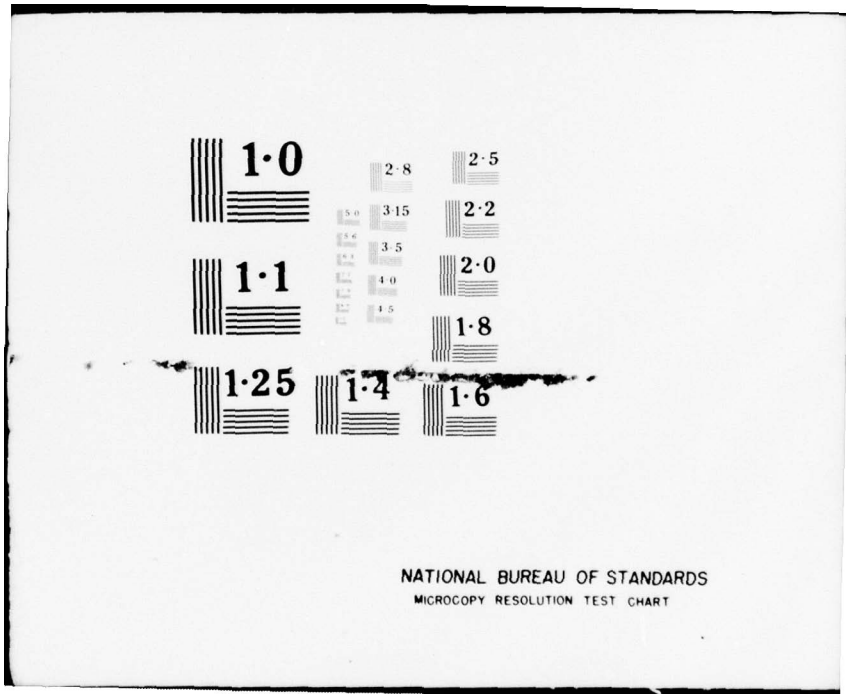
AD-A043 380 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
A FAMILY OF H-GRAPH MODELS FOR SIMPL-T.(U)
JUL 77 S J GORLIN AF-AFOSR-3181-77
UNCLASSIFIED TR-552 AFOSR-TR-77-0948 NL

1 OF 2
AD
A043 380


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

(11)

B.S.

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

See 1473

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-77-0948	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A FAMILY OF H-GRAPH MODELS FOR SIMPL-T.	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
	6. PERFORMING ORG. REPORT NUMBER: TR552
7. AUTHOR(s): Susan J. Gorlin	8. CONTRACT OR GRANT NUMBER(s): AF-AFOSR 77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Maryland, Computer Sciences Department, College Park, Maryland 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB DC 20332	12. REPORT DATE: Jul 77
	13. NUMBER OF PAGES: 96
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this paper is to illustrate the advantage of modeling in the design of a language. An h-graph model is constructed to reflect the basic version of SIMPL-T. As new features are added to the language, the model is extended accordingly.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

409022

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Several h-graph models are presented. The first describes the basic version of SIMPL-T which recognizes 'integer' as its only data type. The model is then extended to reflect the addition of escape mechanisms to SIMPL-T. The EXIT and ABORT statements have been added, whereas the RETURN statement has been assigned an expanded role.

The model is then again extended to reflect the version of SIMPL-T that recognizes 'string' as well as 'integer' data types. Strings are described in two levels of detail: 1) as single units of information, and 2) as lists of individual characters.

Because modeling provides a rigorous definition of the language, the effects of changes made to the language can be detected in the model. New features can be added without sacrificing the reliability of the original language. The family of h-graphs presented in this paper supports these contentions.

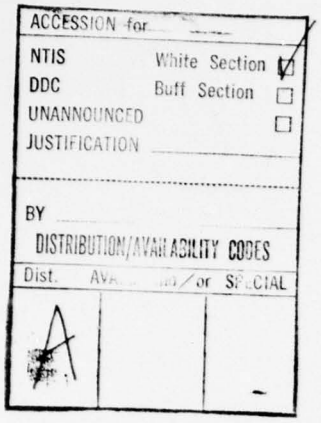
(11)

Technical Report TR-552 July 1977
AFOSR 77-3181

A FAMILY OF H-GRAPH MODELS FOR SIMPL-T*

Susan J. Gorlin

*This research was supported in part by the U.S. Air Force under grant AFOSR 77-3181.



ABSTRACT

The purpose of this paper is to illustrate the advantage of modeling in the design of a language. An h-graph model is constructed to reflect the basic version of SIMPL-T. As new features are added to the language, the model is extended accordingly.

Several h-graph models are presented. The first describes the basic version of SIMPL-T which recognizes 'integer' as its only data type. The model is then extended to reflect the addition of escape mechanisms to SIMPL-T. The EXIT and ABORT statements have been added, whereas the RETURN statement has been assigned an expanded role.

The model is then again extended to reflect the version of SIMPL-T that recognizes 'string' as well as 'integer' data types. Strings are described in two levels of detail: 1) as single units of information, and 2) as lists of individual characters.

Because modeling provides a rigorous definition of the language, the effects of changes made to the language can be detected in the model. New features can be added without sacrificing the reliability of the original language. The family of h-graphs presented in this paper supports these contentions.

TABLE OF CONTENTS

LIST OF TABLES

INTRODUCTION

Semantic modeling provides a set of tools for designing a language in a formal and systematic manner. It provides a rigorous definition of the features of a language that is free of the details of implementation. The model is therefore able to present a high level description of the language which can be of benefit to a wide range of users. The user who is unfamiliar with the language may use the model as a general, yet accurate description of the language, whereas the more experienced user may find it useful as a template for showing how new features might interact within the existing language. The costs involved in making such changes are made clear by the model.

To support these contentions, a family of models will be presented in this paper. The language being modeled is SIMPL-T, a procedure-oriented, structured programming language. The semantic facility chosen is the Hierarchical Graph Language (HGL) [1]. It provides a sufficiently general set of primitives which makes it possible to construct a model that will reflect SIMPL-T as closely as possible. The model must not be so restrictive as to preclude the possibility of being extended for new language features, but at the same time must not be so general as to lose its usefulness as a guide to the specific language in question. The model described in Chapter II of this paper presents a reflection of SIMPL-T which is extended in Chapter III to include escape mechanisms, and in Chapters IV and V to reflect the inclusions of string data in SIMPL-T. Either one, or both of the extensions can be added without compromising the integrity of the original model.

CHAPTER I

HIERARCHICAL GRAPH LANGUAGE (HGL)

At its most basic level, HGL consists of a set of primitives which are defined as a set of nodes and a set of transition functions that change the complexion of these nodes. Each node may be thought of as re-representing a unit of information about the language. Associated with the node may be some atomic data value, some further substructure of information, and some set of attributes. The atomic value (v) can be thought of as representing some program data which may change during execution; the structured value (h) as representing program or data structures; and the attributes (a) as representing some compile-time properties of that node which remain static during execution. These can be defined more formally as mappings. Let

N be a set of nodes

D be a set of atoms

A be a set of attributes and

G be a set of graphs

defined over elements of N. An attributed hierarchical graph (h-graph) over (N,D,A,G) is a 4-tuple (N,v,h,a) where $N \subset N$, $v: N \to D$, $h: N \to G$, and $a: N \times I_k^+ \to A$ where I_k^+ denotes the first k positive integers.

The execution of a program in the model is defined as a sequence of states. Each state is represented by an h-graph which in turn represents the program and its data structures at some point in the execution. An actual state transition is caused by some change in one of the three

mappings v, h, or a on the node, or the inclusion or removal of a node from the set N of nodes. The functions responsible for these changes are:

setv which assigns an atomic value to a node

seth which assigns a graph value to a node

seta which assigns an attribute to a node

define which adds a new node to the nodeset of the h-graph

delete which removes a node from the nodeset of the h-graph.

These functions are more formally defined in Table 1.

It is now the responsibility of the user to define the structures and functions that can be used in conjunction with the HGL primitives to build a model for his language. The data structure is specified by the definition of some particular graph structure over the nodes (e.g., sets, lists, trees, directed graphs) along with a set of construction and accessing primitives that permit the building and traversing of that graph structure. The control structure is specified by the definition of a meta-control language defined over the HGL primitive transition functions and the graph accessing and construction primitives. These basic primitives and structures can now be used as building blocks in the construction of the abstract syntax and semantic functions for the language being modeled.

Before discussing the details of the particular model presented in this paper, a glance at the following figure may help put the various elements of the model into their proper perspective. Each block is built using the definitions of the ones below in conjunction with its adjacent block.

ABSTRACT SYNTAX for L	SEMANTIC FUNCTIONS for L
DATA STRUCTURES	CONTROL STRUCTURES
BASIC HGL PRIMITIVES	

[1]

Two data structures have been chosen for the modeling of SIMPL-T: the directed graph and the list. The directed graph structure is defined as a 4-tuple (N_g, E_g, n_g, e_g) where

N_g is a set of nodes (nodes (g)),

E_g is a set of edges (denoted by arcs (g)),

n_g is a distinguished node called the entry node (denoted by entry (g)),

e_g is the edge label mapping.

The directed graph is used in the compile time environment to create a model which is able to represent the ordering of statements that is implicit in a SIMPL-T program. Its set of arcs, along with its corresponding edge mapping, enable the model to follow a program's flow of control. Once compilation is complete and all statements have been interpreted, the directed graph structure is available for traversal and access. No alteration to these graphs is necessary from this point on. Inasmuch as this paper presents a model that is to be used in the run-time environment when graph construction is already complete, only those primitives that are needed to traverse the graph will be described below:

padj - returns the set of nodes that terminate an outgoing arc from a specified node in the graph

nadj - returns the set of nodes that begin an incoming arc to a specified node in the graph.

Data storage can be handled in a strictly linear fashion. A simplified version of the directed graph which allows each item to point only to its immediate neighbor has therefore been chosen: the list structure. The algorithms generated at execution time follow a strictly linear ordering, and so they too can be handled by this same data structure.

A list is defined by the 4-tuple (N,E,n^f,n^ℓ)

where N is a set of nodes

E is a set of edges defined by the node pairs (n,m), $n,m \in N$ such that each node in N (except for the first node) appears as the first element of a node pair defining an edge exactly once, and each node in N (except for the last node) appears as the second element of a node pair defining an edge exactly once

n^f is the first node (denoted by entry (ℓ))

n^ℓ is the last node.

Lists are both generated and accessed during execution and so the following primitives are defined:

(a) list construction primitives

push which adds a node to the beginning of a list

pop which removes the first node of a list

append which adds a node to the end of a list

list which builds a list from a single node

(b) list accessing primitives

last returns the last node of a list

next returns the next node of a list

item returns a designated node of a list.

A more formal definition of the graph primitives is to be found in TABLE 2, and that of list primitives in TABLE 3, TABLE 4.

The control structures chosen for the modeling of SIMPL-T consist of set expressions, recursive functions, and conditional expressions.

TABLE 1.

HGL STATE TRANSITION PRIMITIVES

Let $S_i = (N_i, v_i, h_i, a_i)$ be the h-graph representation of the program in state S_i and $(N_{i+1}, v_{i+1}, h_{i+1}, a_{i+1})$ be the h-graph representation of the program in state S_{i+1}. The primitives associated with the state transition from S_i to S_{i+1} are defined below:

setv : $N_i \times D \rightarrow N_i$

The execution of setv(n,d) for $n \in N_i$ and $d \in D$ returns the node n and generates the state S_{i+1} whose only new component is v_{i+1} defined by

$$v_{i+1}(m) = \begin{cases} v_i(m), & m \in N_{i+1},\ m \neq n \\ d, & m = n \end{cases}$$

(assigns an atomic value to a node and returns the node)

seth : $N_i \times G \rightarrow N_i$

The execution of seth(n,g) for $n \in N_i$ and $g \in G$ returns the node n and generates the state S_{i+1} whose only new component is h_{i+1} defined by

$$h_{i+1}(m) = \begin{cases} h_i(m), & m \in N_{i+1},\ m \neq n \\ g, & m = n \end{cases}$$

(assigns a graph structure to a node and returns the node)

seta : $N_i \times I_k^+ \times A \rightarrow N_i$

The execution of seta(n,j,p) for $n \in N_i$, $j \in I_k^+$, and $p \in A$ returns the node n and generates the state S_{i+1} whose only new component is a_{i+1} defined by

$$a_{i+1}(m,\ell) = \begin{cases} p, & m = n,\ \ell = j \\ a_i(m,\ell) & \text{otherwise} \end{cases}$$

(seta assigns an attribute to a node and returns the node) [1]

Note: The parameters of the primitive operations setv, seth, seta are evaluated in random order.

define : $N \rightarrow N_i$

The execution of define returns a node $n \in N - N_i$ and generates the state S_{i+1} which is defined in terms of S_i as follows:

$$N_{i+1} = N_i \cup \{n\}$$

$$v_{i+1} = \begin{cases} v_i(m), & m \in N_i \\ \text{undefined}, & m = n \end{cases}$$

$$h_{i+1} = \begin{cases} h_i(m), & m \in N_i \\ \text{undefined}, & m = n \end{cases}$$

$$a_{i+1} = \begin{cases} a_i(m,j), & m \in N_i,\ j \in I_k^+ \\ \text{undefined}, & m = n \end{cases}$$

(define adds a new node to the nodeset of the h-graph)

In addition to this version of the function which appears with no parameters, the following version can be used

define $(n \mid v(n) = d, h(n) = g, a(n,1) = a_1, \ldots, a(n,r) = a_r)$

which combines the addition of the new node to the h-graph along with its associated v, h, a mappings.

delete : $N_i \rightarrow N_i$

The execution of delete(n) returns a node $n \in N_i$ and generates the state S_{i+1} which is defined in terms of S_i as follows:

$$N_{i+1} = N_i - \{n\}$$

$$v_{i+1}(m) = v_i(m),\ \forall\ m \in N_{i+1}$$

$$h_{i+1}(m) = h_i(m),\ \forall\ m \in N_{i+1}$$

$$a_{i+1}(m,k) = a_i(m,k),\ \forall\ m \in N_{i+1}$$

(delete removes a node from the nodeset of the h-graph)

TABLE 2

GRAPH ACCESSING PRIMITIVES

padj : $N \times G \rightarrow 2^N$

The execution of padj(n,g) returns the set of nodes that terminates an outgoing arc from node n in graph g defined by

$$\{m \in nodes(g) \mid (n,m) \in arcs(g)\}$$

nadj : $N \times G \rightarrow 2^N$

The execution of nadj(n,g) returns the set of nodes that begins an incoming arc to node n in graph g defined by

$$\{m \in nodes(g) \mid (m,n) \in arcs(g)\}$$

TABLE 3

LIST CONSTRUCTION PRIMITIVES

push : $L \times N \rightarrow L$

The execution of push(ℓ,n) returns the list ℓ' which is defined by

$$nodes(\ell') = nodes(\ell) \cup \{n\}$$

$$arcs(\ell') = arcs(\ell) \cup \{n, n^f(\ell)\}$$

$$n^f(\ell') = n$$

$$n^\ell(\ell') = n^\ell(\ell)$$

(push adds a node and its associated arc to the beginning of the list and returns the list)

The execution of push(ℓ,n) where ℓ is the empty list, returns the list ℓ' which is defined by

nodes (ℓ') = n

arcs (ℓ') = $\emptyset$

$n^f(\ell')$ = n

$n^\ell(\ell')$ = n

pop : L → L

The execution of pop(ℓ) returns the list ℓ' which is defined by

nodes (ℓ') = nodes (ℓ) - $\{n^f(\ell)\}$

arcs (ℓ') = arcs (ℓ) - $\{(n^f(\ell),m) \mid m \in \text{nodes } (\ell)\}$

$n^f(\ell')$ = $(m \mid m \in \text{nodes } (\ell) \wedge \{n^f(\ell),m)\} \in \text{arcs } (\ell))$

$n^\ell(\ell')$ = $n^\ell(\ell)$

(pop removes the first node of the list along with its associated arc and returns the list)

The execution of pop(ℓ), where ℓ is a list containing one node, returns the list ℓ' which is defined by

nodes (ℓ') = $\emptyset$

arcs (ℓ') = $\emptyset$

$n^f(\ell')$ = $\emptyset$

$n^\ell(\ell')$ = $\emptyset$

append : L x N → L

The execution of append(ℓ,n) returns the list ℓ' which is defined by

nodes (ℓ') = nodes (ℓ) $\cup$ {n}

arcs (ℓ') = arcs (ℓ) $\cup$ $\{n^\ell(\ell),n\}$

$n^f(\ell')$ = $n^f(\ell)$

$n^\ell(\ell')$ = n

(append adds a node and its associated arc to the beginning of the list and returns the list)

The execution of append(ℓ,n) , where ℓ is the empty list, returns the list ℓ' which is defined by

$$\text{nodes}(\ell') = n$$

$$\text{arcs}(\ell') = \emptyset$$

$$n^{f}(\ell') = n$$

$$n^{\ell}(\ell') = n$$

list : $N \rightarrow L$

The execution of list(n) returns the list ℓ' which is defined by

$$\text{nodes}(\ell') = n$$

$$\text{arcs}(\ell') = \emptyset$$

$$n^{f}(\ell') = n$$

$$n^{\ell}(\ell') = n$$

(list builds a list for the node n)

TABLE 4

LIST ACCESSING PRIMITIVES

last : $L \rightarrow n$

The execution of last(ℓ) returns the last node of the list ℓ, i.e., the node which has no outgoing arc emanating from it, defined by

$$n^{\ell}(\ell)$$

next : $N \times L \rightarrow n$

The execution of next(n,ℓ) returns the node of list ℓ which has an incoming arc that emanates from node n, defined by

$$m \mid (n,m) \in \text{arcs}\ (\ell) \wedge n \neq n^{\ell}$$

item : $I \times L \rightarrow n$

The execution of item(i,ℓ) returns the ith node in the list ℓ, defined by

$$\underbrace{\text{next}(\text{next}(\ldots\text{next}(n^{f},\ell),\ldots\ell),\ell)}_{i}$$

CHAPTER II

SIMPL-T (BASIC VERSION)

1. SIMPL-T Syntax

The language being modeled in this section of the paper is the basic version of SIMPL-T. It recognizes data only of type integer. This data is stored either in an integer variable or in a one-dimensional integer array which is defined as an ordered collection of elements. These elements are numbered 0,1,...,n-1 where n is the number of elements in the array, as declared by the user at compile time.

These data configurations appear as terms of a SIMPL-T expression. In its simplest form the expression is a simple variable (e.g., x), or an element of an integer array (e.g., A(i)). It may also take the form of a function (e.g., fact(x)), or a primitive operation (e.g., a+b, a<b, ¬a), or a combination of both. In each of these cases the expression itself represents a value which must be computed by the program at execution time.

The program itself consists of a set of global variables, a set of functions, and a non-empty set of procedures, one of which is designated as the starting procedure. The global variables are integer variables and arrays that are known to all segments (functions and procedures) of the program. If the value of this global variable is not initialized by the program, it is undefined when execution of the program begins.

SIMPL-T is a structured programming language whose flow of control is governed by the execution of its procedures and functions. It is therefore necessary to examine these basic building blocks in some detail.

A procedure is defined by a set of formal parameters, a set of local variables, and a sequence of statements. When the procedure is invoked, the actual parameters supplied by the user are passed to the procedure, and the sequence of statements is executed. Control then returns to the caller. The user may choose to include a RETURN as the last statement in the sequence. It, however, has no effect on the execution of the procedure.

A function is similar to a procedure in that it uses its parameters and local variables, together with the global variables known to the program, to execute its specified sequence of statements. It differs in that upon execution it returns the value of the expression specified in the RETURN statement. This expression and the function itself must both be of type integer. The function RETURN, unlike the procedure RETURN, is a required statement which must appear as the last statement of a function. Both procedures and functions may be recursive.

The execution of a program has been defined as a sequence of states. It is the execution of the statements of a segment that causes these state transitions to be made. A description of the SIMPL-T statements follows:

a) The assignment statement assigns the value of an expression to a variable.

b) The IF statement causes the conditional execution of a sequence of one or more statements. The expression is evaluated at execution time and the appropriate action is determined. If the conditional expression is found to be true ($\neq 0$), the THEN portion of the statement is executed, otherwise execution continues with the ELSE portion (if it exists).

c) The WHILE statement provides a means of writing iterative code. As long as the value of the conditional expression is non-zero, the designated statement list is executed. If its value is zero, execution continues with the statement following the WHILE.

d) The CASE statement provides for the execution of one of a group of statements. The value of the conditional expression is evaluated at execution time and its value is matched against the value assigned to each statement in the group. When a match is found, the corresponding statement is chosen for execution. If no match occurs, the statement referenced by ELSE is chosen (if it exists).

e) The CALL statement causes a specified procedure to be executed. It passes a list of arguments (actual parameters) to the procedure. These arguments must agree in number and type with the formal parameters defined by the procedure itself. Upon completion of the procedure, execution resumes with the statement following the CALL statement.

f) The RETURN statement signifies the end of a procedure or function. In the case of a function, it designates the expression whose value is to be returned.

For a more detailed discussion of the language features SIMPL-T, consult the SIMPL-T manual [3] (pp. 3-17).

2. H-Graph Syntax

With this introduction to SIMPL-T complete, h-graph syntax for the language can now be described. The basic unit of information in the model is a variable. It is represented by a node which has associated with it a value, a graph, and a set of attributes. The attributes remain constant throughout the execution of the program but the value of

the variable may change at any time merely by modifying the v mapping associated with the node.

Variables may be linked together in list form to create a one-dimensional array. This array is represented by a node whose graph consists of nodes representing the individual elements of the array. The number of elements in the array and the type of the array, together with other identifying characteristics are reflected in the attributes associated with the array. At the same time, each element of the array retains its own value, graph, and attributes, as described above.

In addition to the simple variable and simple array just described, the recursive variable and recursive array are necessary for the implementation of this model. Simple variables are joined together in a list structure to form a recursive variable; simple arrays are joined together to form a recursive array. The use of this additional form of data will be made clear when procedures are discussed.

The syntax for an expression is defined either as an operation node or as a node containing a variable node. The operation node in turn translates into an operator followed by a list of operands. The operands are themselves expressions. It is because a variable node eventually appears in the operand list associated with the operator, and because the elements of the operand list must be unique, that we nest the variable node in an additional node. This then permits us, for example, to add the variable x to itself. The graph of the expression would look like [add] → [[x] → [x]] .

The graph of the program node is defined as a set of global variables (i.e., simple variables and simple arrays), a set of functions, and a non-empty set of procedures, one of which is designated as the

entry node of the graph. The procedure node consists of a parameter list, a set of locals, and a statement list. These parameters and local variables describe nodes which pertain only to a particular procedure at a particular point in time. In order to preserve the integrity of the procedure at each nested level of recursion, every one of the variables (both local and parameter) is stored in its own stack as a recursive variable or array. The description of the actual execution of the procedure is reserved for the section which deals with the semantics of the model.

A function is similar to a procedure in that its graph contains a parameter list, a set of locals, and a statement list. In addition, however, its graph contains a node representing the return expression. It is here that the evaluated expression that is returned by the function is to be stored. The evaluated expression and the function itself must both be of type integer.

The list of statements forms the major portion of both the procedure and function nodes. Three of the statements in this model, namely the CALL, assignment, and RETURN statements are treated as operations consisting of a semantic operator followed by a list of operands. The remaining three statements, namely the IF, CASE, and WHILE statements are represented by a node whose graph must be traversed at execution time. The value that results from the evaluation of the entry node of the graph determines the path of traversal.

BNF Notation

The notation used to describe the h-graph syntax is an extension to BNF notation. The following extensions are defined:

a) <a> :: = [<b>] | [<c>]

<a> is defined as (1) [<b>], a node whose h value is given by <b>;

(2) [<c>] , a node whose v value is given by <c>.

b) <f> :: = [<g>]; $\underline{att}_1 = val_1, \ldots, \underline{att}_n = val_n$ | [<d>]

<f> is defined as (1) a node whose h value is given by <g> and whose a_i value is given by val_i for all attributes a_i associated with the node;

(2) a node whose h value is given by <d> and which has no attributes associated with it

(I.e., | has precedence over ;)

c) <h> :: = $\langle i \rangle^{*} \xrightarrow{\langle j \rangle} \langle k \rangle$

<h> is defined as a directed graph from an element of class <i> to an element of class <k> with a directed arc from <i> to <k> whose label is an element of <j> . The entry node of the graph is <i> .

d) <h> :: = $\langle i \rangle^{*} \rightarrow \langle k \rangle$

<h> is defined either as a directed graph or as a list from an element of class <i> to an element of class <k> . The entry node of the graph is <i> .

e) <ℓ> :: = [$\langle m_1 \rangle, \ldots, \langle m_i \rangle^{*}, \ldots, \langle m_n \rangle$]

<ℓ> may be defined as a node whose h value is a graph with n disjoint components from classes $\langle m_1 \rangle$, $\langle m_2 \rangle$,..., $\langle m_n \rangle$. The entry node of the graph is $\langle m_i \rangle$.

It should be noted that the model syntax guarantees the uniqueness of variables passed as arguments to a procedure or function by virtue of the fact that the nodes of a list must be distinct. It is not desirous, however, to place this uniqueness restriction upon the variables appearing in an operand list. For this reason, each variable in the operand list is nested in an expression variable node. Although the expression variables themselves are distinct, the variables nested within them need not be unique.

For example:

Non-unique variable may be added together as follows

[add] → [[x] → [x]]

but unique arguments are passed by a CALL statement as follows

[call] → [PROCA → [x→y]]

Shorthand Notation:

Due to the extensive use of operations throughout this paper, a shorthand form of the h-graph notation has been adopted:

$[\underline{name}] \rightarrow [\langle x_1 \rangle \rightarrow \langle x_2 \rangle \rightarrow \ldots \rightarrow \langle x_n \rangle]$

will be written as

$\underline{name}\ (\langle x_1 \rangle, \langle x_2 \rangle, \ldots, \langle x_n \rangle)$

Some additional abbreviations are defined on p. 30.

Data

Variables

V1 <variable> :: = <simple var> | <rec var>

V2 <simple var> :: = <integer var>

V3 <integer var> :: = [<integer>]; class='data',type='integer',
structure='scalar',form='simple' |

[undefined];class='data',type='integer',

structure='scalar',form='simple'

example

<integer var>

int I [ ] class='data',type='integer',
structure='scalar',form='simple'

V4 <rec var> :: = <rec integer>

V5 <rec integer> :: = [<integer var>$^{*}\{\rightarrow$<integer var>$\}_{o}^{n}$];class='data',

type='integer',structure='scalar',form='rec'

Arrays

A1 <array> :: = <simple array> | <rec array>

A2 <simple array> :: = <integer array>

A3 <integer array> :: = [<integer var>$^{*}\{\rightarrow$<integer var>$\}_{o}^{n}$];class='data',

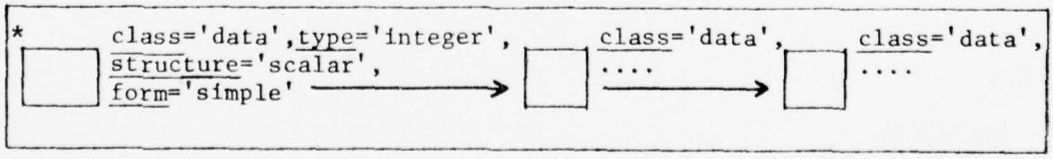
type='integer',structure='array',form='simple',

arraysize=nsize(h(<integer array>))

example

<integer array>

int array A(3)



class='data',type='integer',

structure='array',form='simple'

arraysize=3

A4 <rec array> :: = <rec integer array>

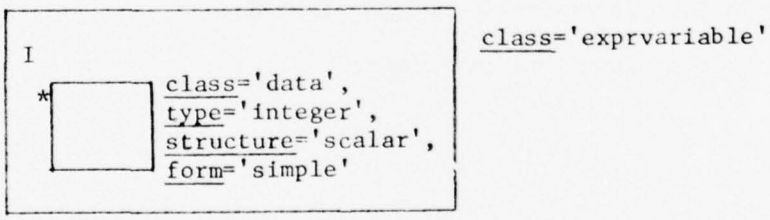
A5 <rec integer array> :: = [<integer array>*{→ <integer array>}$_{o}^{n}$];
class='data',type='integer',structure='array',
form='rec',arraysize=nsize(h(<rec integer
array>))

Expressions

E1 <expr> :: = <operation>|<ref predicate>|[<variable>];
class='exprvariable'
/* A variable is nested within an outer node so that the
variable can be repeated in the operand list - denoted
by class='exprvariable' */

example

(a) <expr>
[<variable>]; class='exprvariable'



Operations

01 <operation> :: = <primitive operation>|<user operation>

02 <primitive operation> :: = [<operator>*→<operand list>];
class='operation'

03 <operand list> :: = [<operand>*{→<operand>}$_{o}^{n}$]

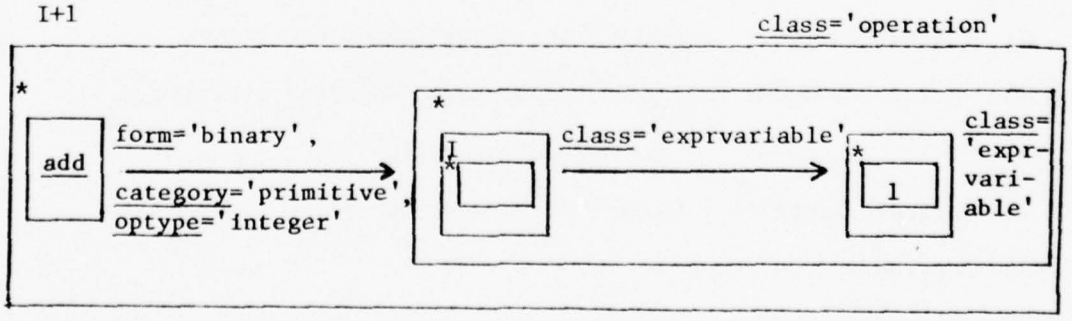
04 <operand> :: = <expr>

```
05  <operator> :: = <primitive binary function>;form='binary',
                       category='primitive'|
                    <primitive unary function>;form='unary',
                       category='primitive'
06  <primitive binary function> :: = <binary integer function>;
                       optype='integer'|
                    <binary relational function>;optype='rel'|
                    <binary logical function>;optype='log'
07  <binary integer function> :: = [add]|[sub]|[mult]| [div]|[ℓℓs]|[ℓcs]|
                    [rℓs]|[ras]|[bitand]|[bitor]|[bitxor]
08  <binary relational function> :: = [ℓt]|[ℓe]|[ge]|[gt]|[eq]|[ne]
09  <binary logical function> :: = [and]|[or]
010 <primitive unary function> :: = <unary integer function>;
                    optype='integer'|
                    <unary logical function>;optype='log'
011 <unary integer function> :: = [minus]|[comp]
012 <unary logical function> :: = [not]
```

example

(b) <expr>

<primitive operation>



013 <user operation> :: = [<user operator>*→<arglist>];class='operation'

014 <user operator> :: = <user function>;category='function'

015 <user function> :: = <func def>

016 <ref predicate> :: = [<ref operation>];class='operation'

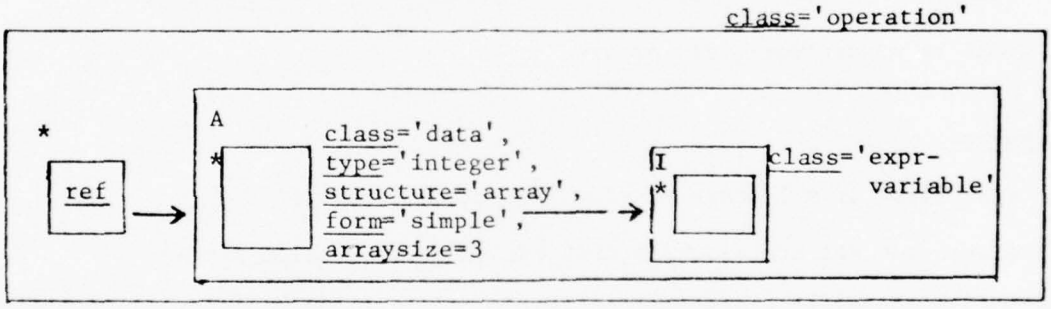
017 <ref operation> :: = ref(<array>,<expr>) /* [ref]→[<array>→<expr>] */

example

(c) <expr>

<ref predicate>

A(I)



where A is defined as an <integer array>

I is defined as an <integer var>

Program Structure

P1 <program> :: = [{<global var>}$_{o}^{n}$,<proc def>*,{<segment def>}$_{o}^{n}$];

class='program'

P2 <global var> :: = <simple var>;scope='global'|<simple array>;

scope='global'

P3 <segment def> :: = <proc def>|<func def>

Procedures

P4 <proc def> :: = [<formal parlist>,<loclist>,<stmt list>*];

class='program',segtype='proc'

Functions

P5 <func def> :: = [<formal parlist>,<loclist>,<stat list>*,<ret var>];
class='program',type=<typename>,segtype='func'

P6 <typename> :: = 'integer'

P7 <ret var> :: = <simple var>;role='return'

P8 <formal parlist> :: = [<par>*{→<par>}$_{o}^{n}$];class='parlist'| [ ];
class='parlist'

P9 <par> :: = <rec var>;calltype=<callname>|<rec array>;
calltype='reference'

P10 <callname> :: = 'value'|'reference'

P11 <loclist> = [<loc>*{→<loc>}$_{o}^{n}$];class='loclist'| [ ];class='loclist'

P12 <loc> :: = <rec var>|<rec array>

Statements

S1 <stat list> :: = [<stat>*{→<stat>}$_{o}^{n}$]; class='program'

S2 <stat> :: = <if stmt>|<while stmt>|<case stmt>|<assign stmt>|
<call stmt>|<return stmt>

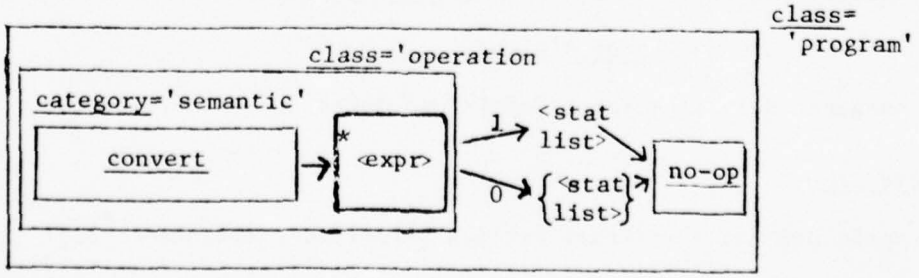
S3 <call stmt> :: = [<call operation>];class='operation'

S4 <assign stmt> :: = [<assign operation>];class='operation'

S5 <if stmt> :: = [<convert predicate>$\overset{*1}{0}$→<stat list>→<end>$\overset{1}{0}$←{<stat list>}];class='program'

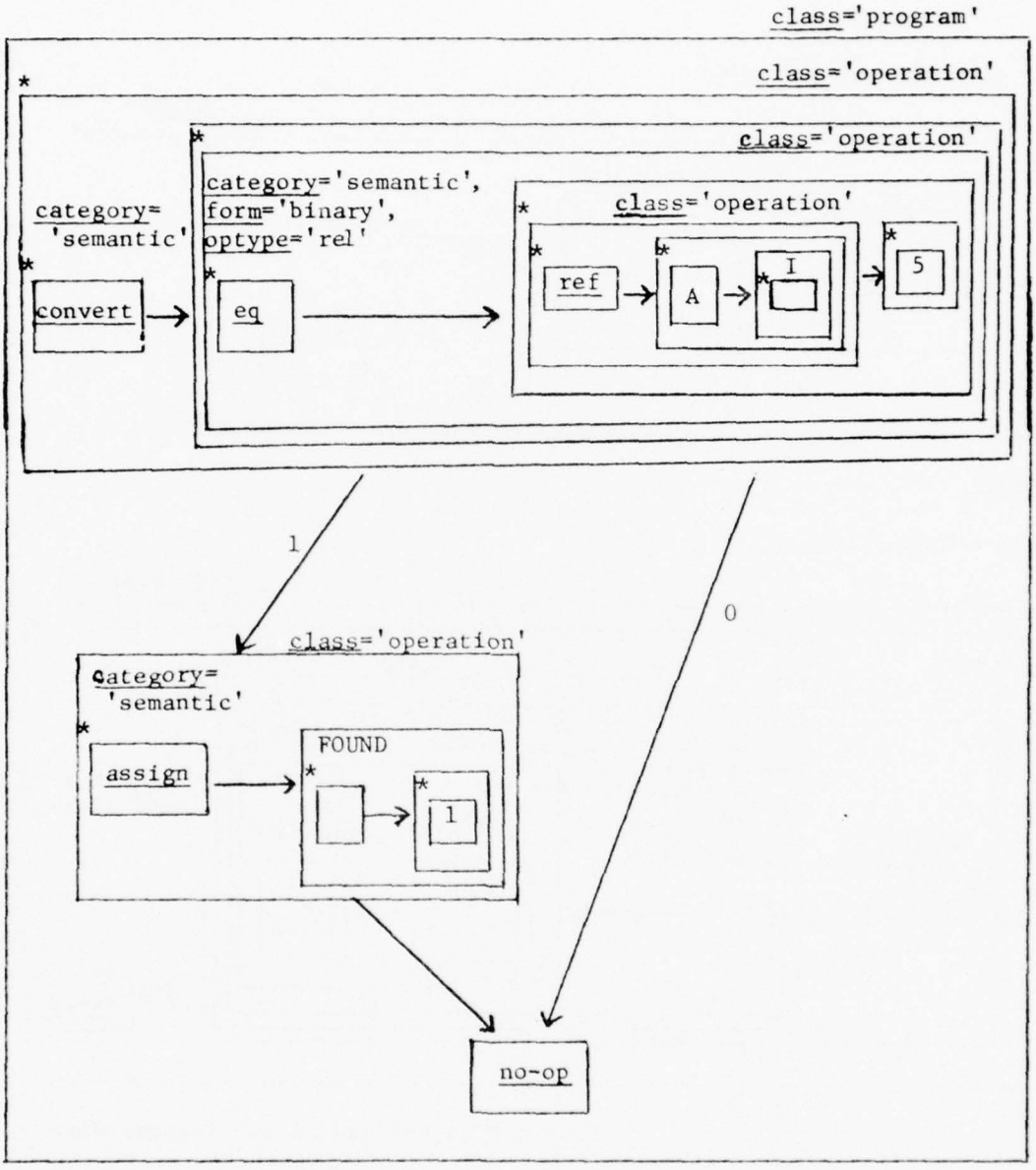
S6 <convert predicate> :: = [<convert operation>];class='operation'

general <if stmt>



example

if A(I) = 5 then FOUND := 1 end



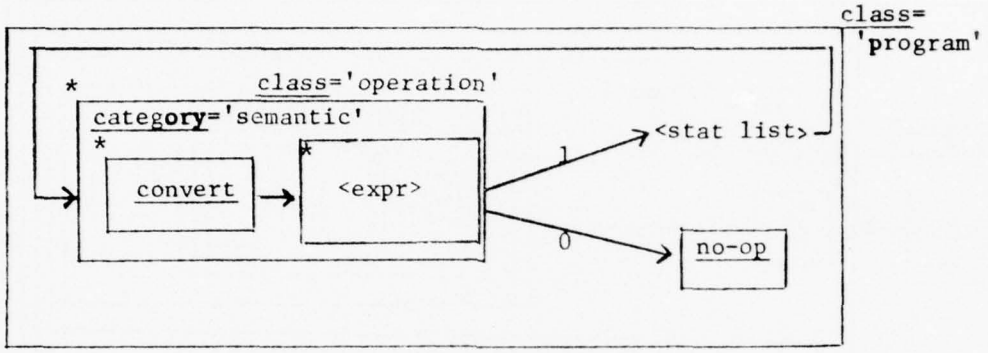
S7 <case stmt> :: = [<case expr> <stat list> <stat list> <stat list> {<stat list>}1 0 <end>] ;class='program'

S8 <case expr> :: = [<case operation>];class='operation'

S9 <case operation> :: = calculate(<expr>) /* [calculate] → [<expr>] */

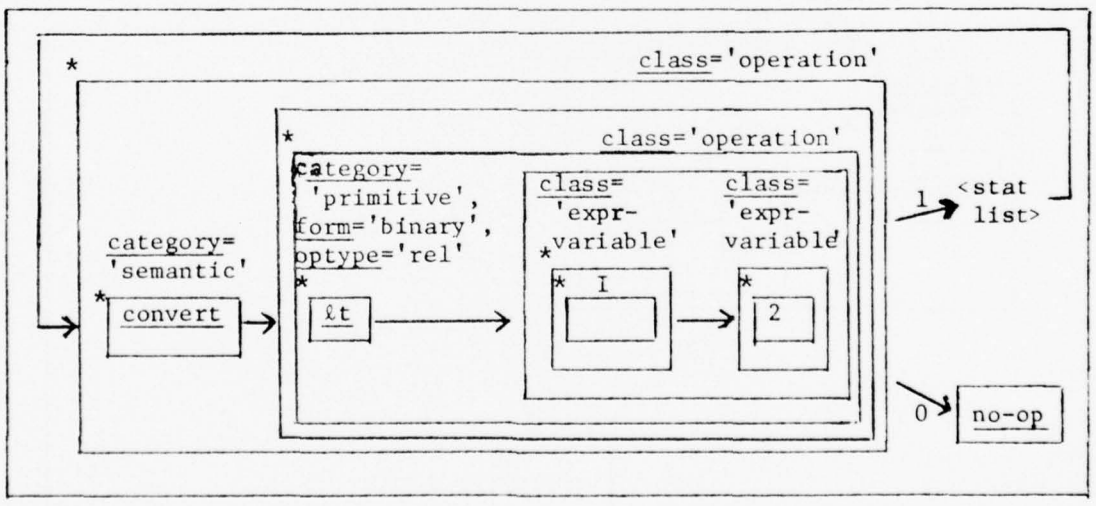
S10 <while stmt> :: = [<convert predicate> *1→ <stat list> ; 0→ <end>] ;class='program'

general <while stmt>



example

while I<2 do <stat list>



where I is defined as an <integer var>

S11 <return stmt> :: = [<return operation>];class='operation' |

[<freturn operation>];class='operation'

S12 <assign operation> :: = assign(<ℓp>,<expr>)

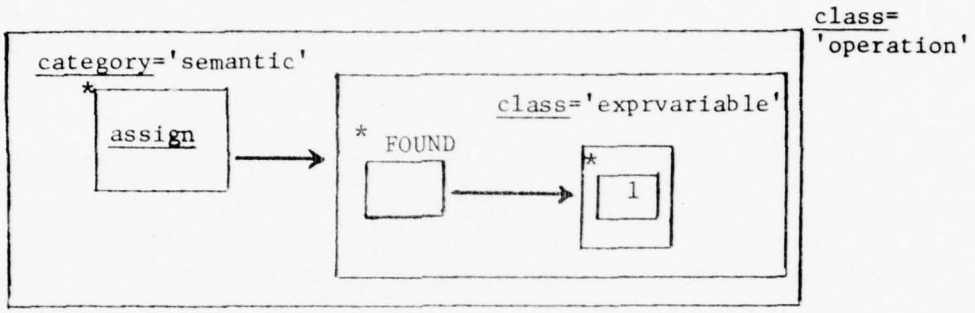
/*i.e., [assign] → [<ℓp> → <expr>] */

S13 <ℓp> :: = <variable>|<ref predicate>

example

<assign stmt>

FOUND : = 1 where FOUND is defined as an <integer var>



S14 <call operation> :: = call(<proc def>,<arglist>)

/* [call] → [<proc def>→<arglist>] */

S15 <arglist> :: = [<arg>* {→<arg>}$_{o}^{n}$] | [ ]

S16 <arg> :: = <array>|<operation>|<ref predicate>|<variable>

/* The arguments in an arglist must be unique, that is, an argument may not appear in the list more than once. Therefore, the variable node need not be reduced another level */

S17 <convert operation> :: = convert(<expr>) /* [convert] → [<expr>] */

S18 <return operation> :: = return( ) /* procedure return

[return] → [ ] */

S19 <freturn operation> :: = freturn(<expr>) /* function return

[freturn] → [<expr>] */

S20 <end> :: = [no-op]

Semantic Operations

F1 <semantic operation> :: = [<semantic operator>* →<semantic operand list>];

class='operation'

F2 <semantic operator> :: = <semantic function>;category='semantic'

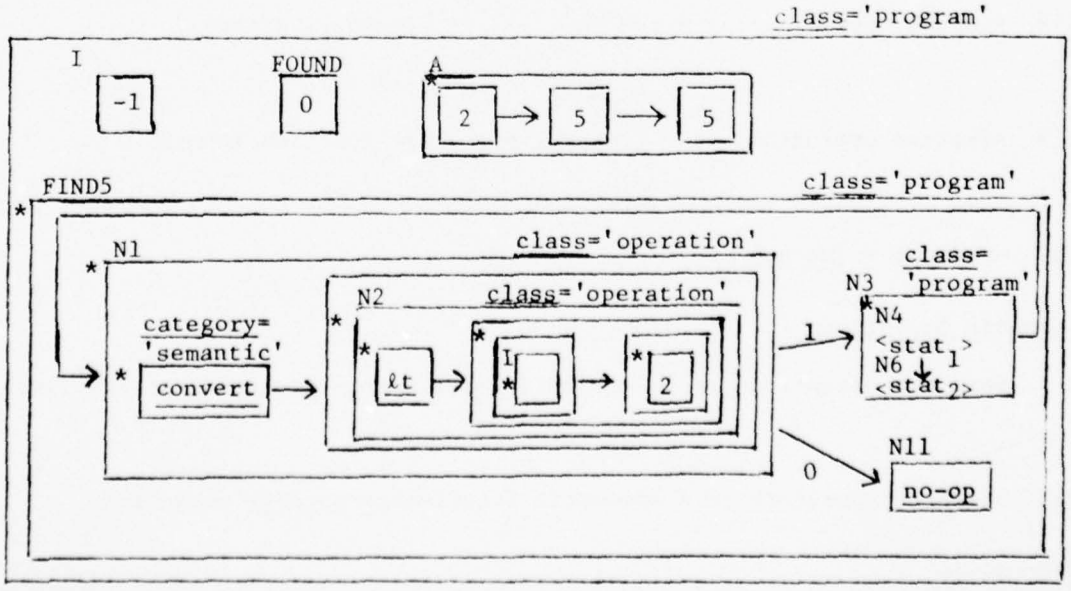
F3 <semantic function> :: = [assign]|[call]|[convert]|[calculate]|
[return]|[freturn]|[eval]|[fcall]|[ref]|[leveleval]|[addlevel]|
[installpar]|[evalargs]|[valuenode]|[refnode]|[buildbuffer]|
[installargs]|[pushpar]|[parattr]|[installocals]|[installoc]|
[pushloc]|[installocarray]|[locattr]|[releaselocs]|[releasepars]|
[release]|[popstack]|[pass]

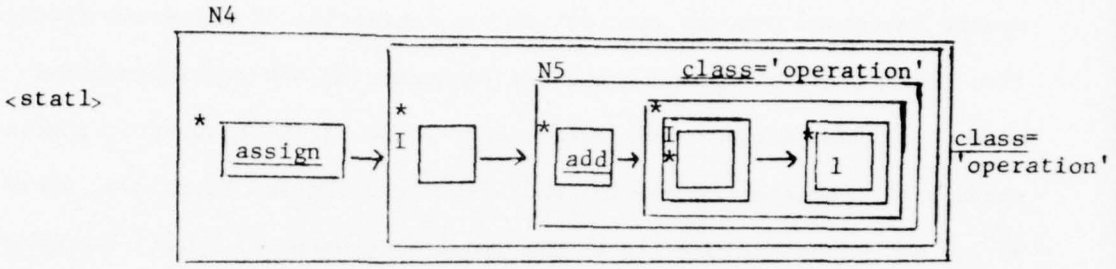
F4 <semantic operand list> :: = [<semantic operand>* {→<semantic operand>}$_{o}^{n}$]
| [ ]

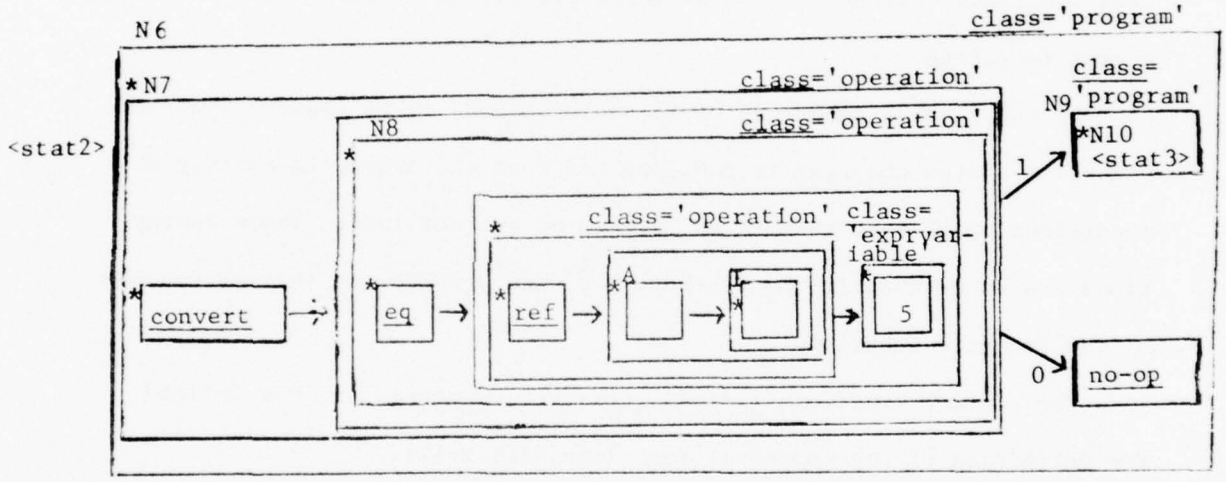
F5 <semantic operand> :: = <expr>|<variable>|<segment def>
<semantic operation>|<formal parlist>|<loclist>|<arglist>

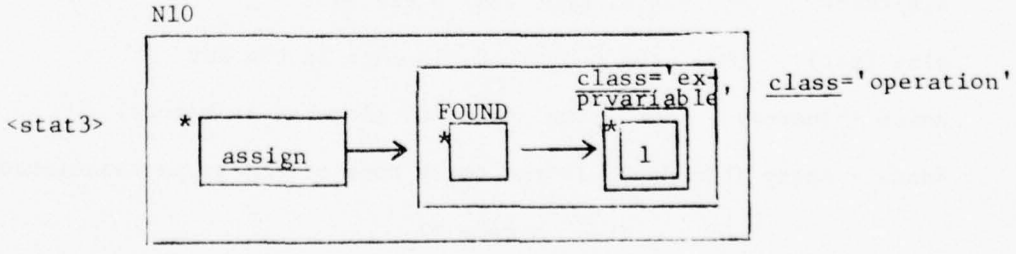
An example of a program which consists of three global variables and a single procedure follows:

```
int I = -1, FOUND = 0
int array A(3) = (2,5,5)
proc FIND5
   while I < 2 do I := I+1
                  if A(I) = 5 then FOUND := 1 end
   end
start FIND5
```




N4
<stat1>
N5
class='operation'
assign
I
add
1
class=
'operation'


N6
class='program'
N7
class='operation'
class=
N9 'program'
N8
class='operation'
<stat2>
N10
<stat3>
class='operation' class=
'expryar-
iable'
1
convert
eq
ref
5
0
no-op


N10
class='ex-
prvariable'
class='operation'
<stat3>
FOUND
assign
1

3. Semantic Functions

The semantics of a programming language are defined as a set of semantic functions over an instruction set consisting of the state transition primitives, the graph structure construction and accessing primitives, and some mathematical primitives. Some of these semantic routines specify the sequence of control and are called control functions, whereas others define the meaning of the language components (e.g., procedures) relative to one another. These are called component functions. The control functions define a mechanism which permits the execution of the component functions.

At the time of execution of these functions, it is assumed that compile-time syntax checking is complete and that the arguments satisfy the conditions which make them valid for use by the routines. These assumptions are enumerated in the brackets $\{\!\{ \}\!\}$ that precede the instruction set of the semantic functions.

The control functions execute, traverse, nextnode are now defined. The definition of the component functions will follow.

Shorthand Notation

The following abbreviations are used by the semantic functions:

```
elt (set)      /*   any element from a set */
size (set)     /*   the number of elements in the set */
nsize (h(node))     /*   the number of elements in h(node) */
*node = entry (h(node))  /* the entry node of the graph associated with
                            a node */
a(node) = {att_i(node)}  /*  the set of attributes of 'node' */
operl(node) = *next(*node,h(node)) /*  the first operand node of the
                        operand list node associated with 'node' */
```

```
oper2(node) = next (oper1(node),h(next(*node,h(node)))  /*  the second
                operand node of the operand list node associated with
                'node'.  */
oper3(node) = next (oper2(node),h(next(*node,h(node)))  /*  the third
                operand node ...  */
oper4(node) = next (oper3(node),h(next(*node,h(node)))  /*  the fourth
                operand node ...  */
pars(segment) = x | x ∈ nodes (h(segment)) ∧ class(x) = 'parlist'
                /* the parameter list of the segment definition
                   'segment'  */
```

CONTROL FUNCTIONS

/* Execute

A single node is passed to this routine and the value resulting from its execution is returned. The manner in which the node is actually executed depends on the composition of the node itself. If the node refers to a single function (in which case its class attribute would be 'operation'), the contents of the node are evaluated by the component function eval and the execution of the node is considered complete. If, however, the node refers to a graph containing a sequence of nodes (in which case its class attribute would be 'program'), additional control functions must be utilized before the execute function is able to return a value. The traverse function provides the means for moving along the graph described by the node to be executed. */

(1) execute (node) =

$$\{\text{class(node)} = \text{'operation'} \vee \text{class(node)} = \text{'program'}\}$$

class(node) = 'operation' ⇒ eval(node)

class(node) = 'program' ⇒ traverse(*node,node)

/* Traverse is a recursive procedure which controls the path of traversal through a graph. It calls for the execution of the node 'current node' (via the execute function) and uses the value of the node returned by its execution to calculate the next node in the graph (via the nextnode function). Traverse then examines this new 'current node'. If it is a data node, we know that the previous node had been the last executable node (see line (c) of nextnode). The new node is returned and the traverse function terminates at this point. If the new node indicates, by way of a message generated by the execution of the previous node, that a function's RETURN statement had been encountered (i.e., if (current node) = freturn), traverse stores the value of the RETURN expression in the graph of the function definition.

If neither of these above conditions has been met, traverse calls itself recursively to calculate the next node in the path and then subjects the new 'current node' to the checks described above. */

(2) traverse (current node, nodeset) =

```
{ class (current node) = 'data' ∨
  (∨ (current node) = freturn ∧ segtype (nodeset) = 'func' ∧
     type (x | role(x)='return' ∧ x ∈ nodes(h(nodeset)))=type
     (*current node) ∧ type (*current node)=type(nodeset))
     /*  type of RETURN expression = type of function */ ∨
  class (current node) = 'program' ∨
  class (current node) = 'operation'

  class (current node) = 'data' ⇒ current node  /*  end of
        traversal  */
  class (current node) = 'return' ⇒
        assign (x | role(x) = 'return' ∧ x ∈ nodes (h(nodeset)),
           current node)
```

/* store value of RETURN expression in the graph of the function definition */

class (current node) = 'program' ∨ class (current node) = 'operation'

⇒ traverse (nextnode (execute (current node), current node, nodeset), nodeset) /* continue traversal */

/* Nextnode examines the previous node ('lastnode') and the node containing the value resulting from the previous node's execution ('lastnodevalue'). If the previous node had been a function RETURN statement (line b), a message to this effect is sent back to traverse. If the previous node had been any other terminal node (line c), the traversal of the graph is complete, and the previous node is passed back to the traverse function. If the last node executed had not been a terminal node and had had one arc directed to the next node (line d), this new node is returned to traverse. If, however, more than one arc had emanated from the last node executed, the arc label must be matched against the value resulting from the execution of the previous node (line e,f). When a unique match is found, the next node to be examined is returned to traverse. */

(3) nextnode (lastnodevalue, lastnode, nodeset) =

∨ (lastnodevalue) = undefined ∨

∨ (lastnodevalue) = freturn ∧ size (padj(lastnode, h(nodeset))) = 0 ∨

/* RETURN appears as last statement in the graph */

0 ≤ size (padj(lastnode,h(nodeset))) ≤ 1

/* 0 or 1 successors */ ∨

((∃x) x ∈ padj (lastnode,h(nodeset)) ∧ e (lastnode,x)= h(nodeset)

∨ (lastnodevalue) /*matching arc label for IF, WHILE, CASE statements*/ ∨

$$(\exists x)\ x \in padj(lastnode, h(nodeset)) \wedge e(lastnode, x) = 'else')$$
$$\wedge\ size(padj(lastnode, h(nodeset)) \mid e(lastnode, x) = 'else') = 1)$$
$$\nu(lastnodevalue) \quad \text{/*only one match exists*/}$$

/* (a) pass on error message */

ν (lastnodevalue) = undefined $\Rightarrow$ lastnodevalue

/* (b) pass on return message */

ν (lastnodevalue) = freturn $\Rightarrow$ lastnodevalue

/* (c) no arcs emanating from lastnode, indicating that it was the last executable node in the graph - pass on its value */

size (padj(lastnode,h(nodeset)))=0 $\Rightarrow$ define (x|class(x)='data', $\nu(x)=\nu$(lastnodevalue))

/* (d) return the one successor to lastnode in the graph */

size (padj(lastnode,h(nodeset))) = 1 $\Rightarrow$ elt(padj(lastnode, h(nodeset)))

/* (e) choose the appropriate node from among the multiple successors to lastnode (applies to IF, WHILE, CASE statements). ι (iota) denotes a 'choose' function which selects the unique value for which the condition is true. */

$(\exists x)(x \in padj(lastnode, h(nodeset)) \wedge e(lastnode, x) = \nu(lastnodevalue)) \Rightarrow$

$(\iota x)(x \in padj(lastnode, h(nodeset)) \wedge e(lastnode, x) = \nu(lastnodevalue))$

/* (f) choose the node from among the successors to lastnode whose arc label is 'else' (applies to CASE statement) */

$(\exists x)\ (x \in padj(lastnode, h(nodeset)) \wedge e(lastnode, x) = 'else') \Rightarrow$

```
(ιx) (x∈ padj (lastnode,h(nodeset)) ∧ e (lastnode,x) =
                                                h(nodeset)
    'else')
```

An example might be of help in understanding the flow of control dictated by the functions defined above. Using the program described on page 28 for this purpose, the steps generated by the execute command will be described in detail, beginning at the point where I = 1 .

The procedure FIND5 is examined by the execute routine and is found to have class = 'program' . The nodes of the procedure must therefore be traversed, beginning with N1, the WHILE statement. The conditional expression (I < 2) is evaluated and is found to be true (i.e., 1). The nextnode routine matches this value 1 against the edge labels emanating from N1 and determines that the next node to be traversed must be N3, a statement list. This statement list must itself be traversed, beginning with the assignment statement in N4. Upon completion of the assignment, nextnode determines the next node in the path of traversal. Only one arc emanates from N4 and so the next node must therefore be N6, the IF statement. The conditional expression (A(I) = 5) is evaluated and is found to be true (1). By matching the edge labels against this value, nextnode determines that N9, the statement list associated with the IF is next. The assignment is made and nextnode again is called to find the next step. No new node can be found and a data node is returned to signal the end of the statement list contained in N9. Likewise, no new node can be found to follow N9 and the data node is returned to signal the end of the IF statement in N6. Again nextnode is called to continue the traversal of the statement list associated with the WHILE statement (N3), but no node can be found to follow N6. The traversal of N3 is said to be complete. One arc emanates from this node,

and nextnode therefore chooses N1 to be the next node in the path. The WHILE loop will be entered once again.

The value of I is now 2, however, and the conditional expression (I < 2) is found to be false (i.e., 0). The next node in the path is determined to be N11 by the nextnode routine. Traverse returns a data node, and the execution of the procedure FIND5 is complete.

Example

where I=1

P refers to the procedure FIND5

(1) refers to a node whose value is 1, (0) to a node whose value is 0

((1)) refers to the expression variable whose variable node has a value of 1

FOUND data refers to a data node whose value is the value of FOUND

ex = execute
tr = traverse
nn = nextnode

ex P = tr(N1,P)

= tr(nn(ex(N1),N1,P),P)

eval(N1)

convert(N2)

nn((1),N1,P)

= tr(N3,P)

= tr(nn(ex(N3),N3,P),P)

tr(N4,N3)

tr(nn(ex(N4),N4,N3),N3)

eval(N4)

assign(I,N5)

tr(nn(I,N4,N3),N3)

tr(N6,N3)

tr(nn(ex(N6),N6,N3),N3)

tr(N7,N6)

tr(nn(ex(N7),N7,N6),N6)

eval(N7)

convert(N8)

tr(nn(1,N7,N6),N6)

tr((N9,N6)

tr(nn(ex(N9),N9,N6),N6)

tr(N10,N9)

tr(nn(ex(N10),N10,N9),N9)

eval(N10)

assign(FOUND 1)

tr(nn(FOUND, N10,N9),N9)

tr(FOUND data, N9)

tr(nn(FOUND data, N9,N6),N6)

tr(FOUND data,N6)

tr(nn(FOUND data,N6,N3),N3)

tr(FOUND data,N3)

tr(nn(FOUND data,N3,P),P)

= tr(N1,P)

I=2

= tr(nn(ex(N1),N1,P),P)

eval(N1)

convert(N2)

= tr(nn(0,N1,P)

= tr(N11,P)

= N11

COMPONENT FUNCTIONS

/* Eval

A single parameter (node) is passed to this routine for evaluation. If the node contains data, the node is returned and the evaluation is complete. If the node is an expression variable (i.e., class(node) = 'exprvariable'), it is evaluated by an auxiliary routine leveleval which examines the inner node for data. If the node represents an operation, the action taken depends on the type of operation specified. An operator which is in a category called 'primitive' indicates that some basic integer (comp, add) or relational/logical (eq, not, and) operation is to be performed on some integer expression(s). The node whose operator is of the category 'function' is directed to the fcall routine for further evaluation. The fcall routine calculates the value of the expression designated in the RETURN statement and passes the node containing this value (albeit encased in an additional node) to the eval routine. The eval routine, in turn, passes this node to its user and the evaluation is complete.

All other operations passed to this routine involve the evaluation of semantic functions. The value of the operator determines which of the routines described below is to be executed. The node returned by the executing routine is returned to eval which then returns it to its user.

(4) eval (node) =

class(node)='data' ∧ (form(node)='rec' ∨ form(node)='simple') ∨
class(node)='exprvariable' ∨
class(node)='operation' ∧ ((category(*node)='primitive'
∧ ((form(*node)='binary'
∧ (optype(*node)='integer' ∨ optype(*node)=
'rel' ∨ optype(*node)='log')
∧ integerexpr(operl(node))=1 ∧ integerexpr(oper2
(node))=1)
∨ (form(*node)='unary'
∧ (optype(*node)='integer' ∨ optype(*node)='log')
∧ integerexpr(operl(node))=1)))
∨ category(*node)='function'
∨ (category(*node)='semantic' ∧ (ν(*node)=
assign ∨ ν(*node)=convert ∨
ν(*node)=calculate ∨ ν(*node)=call ∨
ν(*node)=ref ∨ ν(*node)=return ∨
ν(*node)=freturn ∨ ν(*node)=
leveleval ∨ ν(*node)=addlevel ∨
ν(*node)=installpar ∨ ν(*node)=
evalargs ∨ ν(*node)=valuenode ∨
ν(*node)=refnode ∨ ν(*node)=build-
buffer ∨ ν(*node)=installargs ∨
ν(*node)=pushpar ∨ ν(*node)=
parattr ∨ ν(*node)=installocals ∨
ν(*node)=installoc ∨ ν(*node)=
pushloc ∨ ν(*node)=installoc-
array ∨ ν(*node)=

locattr ∨ ν(*node)=releaselocs ∨
ν(*node)=releasepars ∨ ν(*node)=
release ∨ ν(*node)=popstack ∨
ν(*node)=pass)))


```
/* node to be evaluated is a data node */
   class(node)='data' ⇒
        form(node)='rec' ⇒ *node
        form(node)='simple' ⇒ node
/* node to be evaluated is a variable node */
   class(node)='exprvariable' ⇒ leveleval(node)
/* node to be evaluated represents an operation */
   class(node)='operation' ⇒
        category(*node)='primitive' ⇒
                form(*node)='binary' ⇒
                    ν(*node)(eval(oper1(node)),eval(oper2(node)))
                    /* where if ν(*node)=add ⇒ add(eval(oper1(node)),
                                                   eval(oper2(node)))
                                ν(*node)=sub ⇒ sub(eval(oper1(node)),
                                                   eval(oper2(node)))
                                etc. */
                form(*node)='unary' ⇒
                    ν(*node)(eval(oper1(node)))
/* node to be evaluated represents a function call */
      category(*node)='function' ⇒ fcall(*node,padj(*node),pars(*node))
/* node to be evaluated represents a semantic function */
      category(*node)='semantic' ⇒
              ν(*node)=assign ⇒ assign(oper1(node),oper2(node))
```

```
ν(*node)=convert ⇒ convert(oper1(node))

ν(*node)=calculate ⇒ calculate(oper1(node))

ν(*node)=call ⇒ call(oper1(node),oper2(node),
          pars(oper1(node)))

ν(*node)=ref ⇒ ref(oper1(node),eval(oper2(node)))

ν(*node)=return ⇒ return([ ])

ν(*node)=freturn ⇒ freturn(oper1(node))

ν(*node)=leveleval ⇒ leveleval(oper1(node))

ν(*node)=addlevel ⇒ addlevel(oper1(node))

ν(*node)=installpar ⇒ installpar(oper1(node),oper2(node))

ν(*node)=evalargs ⇒ evalargs(oper1(node),oper2(node),
            oper3(node),oper4(node))

ν(*node)=valuenode ⇒ valuenode(oper1(node))

ν(*node)=refnode ⇒ refnode(oper1(node))

ν(*node)=buildbuffer ⇒ buildbuffer(oper1(node),
            oper2(node))

ν(*node)=installargs ⇒ installargs(oper1(node),
            oper2(node),oper3(node),oper4(node))

ν(*node)=pushpar ⇒ pushpar(oper1(node),oper2(node))

ν(*node)=parattr ⇒ parattr(oper1(node))

ν(*node)=installocals ⇒ installocals(oper1(node))

ν(*node)=installoc ⇒ installoc(oper1(node),oper2(node))

ν(*node)=pushloc ⇒ pushloc(oper1(node))

ν(*node)=installocarray ⇒ installocarray(oper1(node),
            oper2(node),oper3(node))

ν(*node)=locattr ⇒ locattr(oper1(node))

ν(*node)=releaselocs ⇒ releaselocs(oper1(node))
```

```
        ν(*node)=releasepars ⇒ releasepars(oper1(node))
        ν(*node)=release ⇒ release(oper1(node),oper2(node))
        ν(*node)=popstack ⇒ popstack(oper1(node))
        ν(*node)=pass ⇒ pass(oper1(node))
```

/* This routine examines the entry of the expression variable 'node' for data. It returns a data node nested in an outer node (i.e., expression variable node). */

(5) leveleval (node) =

```
        class(node)='exprvariable' ∧
        class(*node)='data' ∧ (form(*node)='simple' ∨
              form(*node)='rec')
        form(*node)='simple' ⇒ node     /* return expression variable*/
        form(*node)='rec' ⇒ addlevel(**node)
                                        /* choose entry of recursive
                                           variable and nest it in an
                                           expression variable */
```

/* This routine guarantees that an expression is returned by building a new node around the original node passed as a parameter. */

(6) addlevel (node) =

```
        define (x | class(x) = 'exprvariable', h(x) = list(node))
```

/* Assign evaluates the right part 'rp' of the statement and assigns its value to the node represented by 'ℓp'. Being that 'rp' signifies an expression, the inner node's value (namely ν(*eval(rp))) is used in the assignment. */

(7) assign (ℓp,rp) =

```
        /* assignment to integer variable or integer array element by
```

an integer expression */

{ (integervariable(ℓp)=1 ∨ integerarrayelement(ℓp)=1)
∧ integerexpr(rp)=1 }

integervariable(ℓp)=1 ⇒ setv(eval(ℓp),ν(*eval(rp)))

integerarrayelement(ℓp)=1 ⇒ setv(*eval(ℓp),ν(*eval(rp)))

/* ref returns an expression variable whose inner
node is assigned a value */

/* Ref uses the evaluated expression ('var') as an index into 'array'. It calls addlevel to encase the desired array element in an expression variable */

(8) ref (array,var) =

{ type(*var)='integer' /* index into array is an integer */
∧ structure(array)='array' ∧ (form(array)='rec'∨
form(array)='simple') }

0 ≤ ν(*var) < arraysize(array) ⇒ /* element lies within
bounds of array */

form(array)='rec' ⇒ addlevel(item(ν(*var) + 1, h(*array)))

form(array)='simple' ⇒ addlevel(item(ν(*var) + 1, h(array)))

T ⇒ addlevel(define(x|ν(x)=undefined,class(x)='data')) /* error
conditon */

/* Convert evaluates the expression contained in 'node' and returns a node whose value is 1 if the condition is satisfied (i.e., evaluates to non-zero), and 0 otherwise */

(9) convert (node) =

{ (node)=<expr> }

ν(*eval(node))=0 ⇒ define(x|ν(x)=0)

ν(*eval(node))≠0 ⇒ define(x|ν(x)=1)

/* Calculate evaluates the expression contained in 'node' and returns a node whose value is that of the evaluated expression. */

(10) calculate (node) =

≀ (node)=<expr> ≀

define (x|ν(x)=ν(*eval(node)))

/* The parameter list for a procedure return is empty. Being that the RETURN statement is the last statement of the procedure, it does not affect the flow of control and no action need be taken. */

(11) return (node) =

≀ (node)= [ ] ≀

node

/* The parameter list for a function freturn represents a variable expression. A message node is returned whose value is freturn and whose graph corresponds to that of the evaluated expression */

(12) freturn (node) =

≀ (node)=<expr> ≀

define (x|ν(x)=freturn, h(x)=h(eval(node)))

/* This routine returns the value 1 if 'expr' is an integer expression and 0 otherwise. */

(13) integerexpr (expr) =

class(expr)='operation' ∧ (optype(*expr)='integer'

∨ optype(*expr)='rel' ∨ optype(*expr)='log')

∨ type(*expr)='integer' ⇒ 1

T ⇒ 0

```
/* This routine returns the value 1 if 'node' is an integer array
 element and 0 otherwise. */

(14) integerarrayelement (node) =
        class(node)='operation' ∧ ∨(*node)=ref ∧ type(oper 1(node)) =
                'integer' ⇒ 1
     •  T                 ⇒ 0

/* This routine returns the value 1 if 'node' is an integer variable,
 and 0 otherwise */

(15) integervariable (node) =
        structure(node)='scalar' ∧ type(node)='integer' ⇒ 1
        T                                               ⇒ 0

/* Call executes the subprogram which is represented by the list:
     [installpar(args,pars)]* → [installocals(proc)] → proc →
     [releaselocs(proc)] → [releasepars(proc)]
   The list primitive append is used to construct the list */

(16) call (proc,args,pars) =
        proc=<proc def> ∧
        args=<arglist> ∧
        pars=<formal parlist> ∧ pars ∈ nodes(h(proc))
                ∧ class(pars)='parlist'
     execute (define(x|class(x)='program', h(x)=
        append(append(append(append(list([installpar(args,pars)]),
          [installocals(proc)]),proc), [releaselocs(proc)]),
            [releasepars(proc)])))

/* The function call routine fcall executes the subprogram which is
 represented by the list:
```

```
[installpar(args,pars)]* → [installocals(func)] → func →
    [releaselocs(func)] → [releasepars(func)] → [pass(x|role(x)=
        'return' ∧ x ∈ nodes(h(func)))] */
```

(17) fcall (func,args,pars) =

```
{ func=<func def> ∧
  args = <arglist> ∧
  pars= <formal parlist> ∧ pars ∈ nodes(h(func)) ∧ class(pars)=
            'parlist' }

execute (define(x|class(x)='program', h(x)=
    append(append(append(append(append(list([installpar(args,pars)]),
        [installocals(func)]),func),[releaselocs(func)]),
            [releasepars(func)]),[pass(x|role(x)='return'∧ x ∈
            nodes(h(func)))]])))
```

/* The following routines assign the arguments ('args') supplied by the calling routine to the parameters ('pars') declared in the segment definition. Each parameter in turn is examined and if it is to be passed by value, the corresponding argument is evaluated and its value is stored in a buffer. Otherwise, the graph of the corresponding argument is stored in the buffer. When the parameter list has been exhausted, each parameter is again examined in succession. If the parameter is to be passed by value (i.e., calltype = 'value'), the value stored in the buffer is assigned to its corresponding parameter. If the paramter is to be passed by reference (i.e., calltype = 'reference'), the graph of the argument is assigned to the corresponding parameter. This process continues until all parameters have been assigned.

Upon entry to these routines it is assumed that the number of parameters is equal to the number of arguments, and that each evaluated argument agrees in type and structure to its corresponding parameter.

Installpar initializes the buffer to the empty node, the index to 1, and then calls evalargs. */

(18) installpar (args,pars) =

evalargs(define,define(**x**|ν(x)=1),args,pars)

/* Evalargs stores the evaluated argument in a buffer for each parameter passed by value, and stores the graph of the argument in a buffer for each parameter passed by reference */

(19) evalargs (buffer,index,args,pars) =

nsize(h(args))=nsize(h(pars)) ∧
structure(item(ν(index),h(pars)))=structure(eval(item(ν(index),
h(args)))) ∧
type(item(ν(index),h(pars)))=type(eval(item(ν(index),
h(args)))) ∧
form(item(ν(index),h(pars)))='rec' ∧ class(item(ν(index),
h(pars)))='data' ∧
(form(eval(item(ν(index),h(args))))='simple' ∨
form(eval(item(ν(index),h(args))))='rec') ∧
((calltype(item(ν(index),h(pars)))='value' ∧
((class(item(ν(index),h(args)))='data'
∧ class(item(ν(index),h(args)))='scalar')
∨ class(item(ν(index),h(args)))='operation')) ∨
(calltype(item(ν(index),h(pars)))='reference' ∧
(class(item(ν(index),h(args)))='data'

```
                ∨ (class(item(ν(index),h(args)))='operation'∧
                      ν(*item(ν(index),h(args)))='ref'))))
  ν(index) > nsize(h(args)) ⇒ installargs(buffer,setv(index,1),
              args,pars)      /* buffer construction is complete */
  ν(index) ≤ nsize(h(args)) ⇒
       /* parameter passed by value */
      calltype(item(ν(index),h(pars)))='value' ⇒
       /* evaluate argument, store its value in buffer, and then
            proceed to process next argument */
         evalargs(buildbuffer(buffer,valuenode(eval(item(ν(index),
              h(args))))),setv(index,ν(index)+1), /* increment index */
              args,pars)
       /* parameter passed by reference */
        calltype(item(ν(index),h(pars)))='reference' ⇒
       /* evaluate argument, store node in buffer, and then proceed
            to process next argument */
             evalargs(buildbuffer(buffer,refnode(eval(item(ν(index,
                  h(args))))),
              setv(index,ν(index)+1),  /* increment index */
                 args,pars)
/*  Valuenode returns a node whose value is equal to that of an evaluated
 argument 'node' */
(20)  valuenode (node) =
      (class(node)='exprvariable' ∧ type(*node)='integer') ∨
      (class(node)='data'    ∧ type(node)='integer')
     class(node)='exprvariable' ⇒
         type(*node)='integer' ⇒ define(x|ν(x)=ν(*node))
```

```
        class(node)='data' ⇒
            type(node)='integer' ⇒ define(x|ν(x)=ν(node))
/* Refnode returns the evaluated node, stripping the outer node of an
 expression variable, if necessary */
(21)  refnode (node) =
        { class(node)='exprvariable' ∨ class(node)='data' }
        class(node)='exprvariable' ⇒ *node
        class(node)='data' ⇒ node
/* Buildbuffer adds a node to the buffer and returns the buffer */
(22)  buildbuffer (buffer,node) =
        seth(buffer,append(h(buffer),node))
/* Installargs pops the buffer containing the evaluated arguments and
 assigns these arguments to their appropriate parameters. */
(23)  installargs (buffer,index,args,pars) =
        ν(index) > nsize(h(args)) ⇒ define(x|ν(x)=true)  /* parameter
                                    installation complete */
        ν(index) ≤ nsize(h(args)) ⇒
          execute(define (x|class(x)='program',h(x)=
            append(list([pushpar(item(ν(index),h(pars)),*buffer)]),
              [installargs(seth(buffer,pop(h(buffer))),
                           setv(index,ν(index)+1),args,pars)])))
                /* pop buffer, increment index, then proceed to
                   install next parameter */
/* Pushpar adds the evaluated argument onto the beginning of the appro-
priate parameter list. */
(24)  pushpar (parameter,buffernode) =
        parattr(seth(parameter,push(h(parameter),buffernode)))
```

```
/* Parattr assigns the attribute of the previous first node to the cur-
 rent first node */
(25) parattr (parameter) =
        seta(*parameter,a(next(*parameter,h(parameter))))
/* The following routines add a variable to the list associated with
 each scalar local variable, and add an array to the list associated
 with each local array variable.  The value of the local is undefined
 upon entry into the segment.
     Installocals initializes index to 1, and calls installoc. */
(26) installocals (segment) =
        segment= <segment def> ∧
        locs= <loclist> ∧ locs ∈ nodes(h(segment)) ∧
             class(locs)='loclist'
       installoc(define(x|ν(x)=1),locs)
(27) installoc (index,locs) =
        (structure(item(ν(index),h(locs)))='scalar' ∨
            structure(item(ν(index),h(locs)))='array') ∧
        form(item(ν(index),h(locs)))='rec' ∧
        type(item(ν(index),h(locs)))='integer'
      ν(index) > nsize(h(locs)) ⇒ define(x|ν(x)=true) /* installation
                                                          of locals
                                                          complete */
      ν(index) ≤ nsize(h(locs)) ⇒
          structure(item(ν(index),h(locs)))='scalar' ⇒
             execute(define(x|class(x)='program',h(x)=
                append(list([pushloc(item(ν(index),h(locs)))]),/*install
                                                          variable*/
```

```
                    [installoc(setv(index,v(index)+1),locs)])))
                /* proceed to process next local */
            structure(item(v(index),h(locs)))='array' ⇒
              execute(define(x|class(x)='program',h(x)=
                append(list([installocarray(define,define(x|v(x)=
                  arraysize(item(v(index),h(locs)))),item(v(index),
                      h(locs)))]),          /* install array */
                [installoc(setv(index,v(index)+1),locs)])))
                  /* proceed to process next local */
/* Pushloc adds a local variable to its corresponding stack.  The stack
 is initially set to a single node with attributes. */
(28) pushloc (local) =
      ≀ structure(local)='scalar' ∧ form(local)='rec' ∧          ≀
              type(local)='integer'
          type(local)='integer' ⇒
              seth(local,push(h(local),define(x|a(x)=a(*local))))
/* Installocarray builds an array and then adds it to its corresponding
 stack.  (The stack is a recursive array which is initially set to an
 array with attributes.)  Array is initially set to  [ ]  and v(count)
 is set to the size of the array.  Count is decremented until the array
 has been completed.  The push primitive is used to install the array on
 its stack. */
(29) installocarray (array,count,stack) =
      ≀ type(stack)='integer' ∧ form(stack)='rec'                 ≀
      v(count)=0 ⇒          /* array ready to be added to stack */
          locattr(seth(stack,push(h(stack),array)))
      v(count)≠0 ⇒          /* array building still in progress */
```

```
        type(stack)='integer' ⇒
      installocarray(seth(array,push(h(array),define(y|a(y)=
                        a(*(*stack))))),setv(count,v(count)-1),
                        stack)
          /* proceed to install next element of array */
/* Locattr assigns the attribute of the previous first array to the
current first array */
(30) locattr (stack) =
        seta(*stack,a(next(*stack,h(stack))))
/* The following routines release the locals and parameters. Releaselocs
calls on release to release locals. Releasepars calls on release to re-
lease parameters. */
(31) releaselocs (segment) =
     { segment= <segment def> ∧                                          }
     { locs= <loclist> ∧ locs ∈ nodes(h(segment)) ∧ class(locs)=         }
     {                          'loclist'                                }
     release(define(x|v(x)=1),locs)
(32) releasepars (segment) =
     { segment= <segment def> ∧                                          }
     { pars= <formal parlist> ∧ pars ∈ nodes(h(segment)) ∧               }
     {                          class(pars)='parlist'                    }
     release(define(x|v(x)=1),pars)
/* Release is used to release both locals and parameters. The value of
 index is initially set to 1. It calls on popstack to pop the local or
 parameter */
(33) release (index,locsorpars) =
        v(index) > nsize(h(locsorpars)) ⇒ define(x|v(x)=true)/*release
                                                    is complete*/
```

```
        v(index) ≤ nsize(h(locsorpars)) ⇒
            execute(define(x|class(x)='program',h(x)=
                append(list([popstack(item(v(index),h(locsorpars)))]),
                    [release(setv(index,v(index)+1),locsorpars)])))
                /* proceed to release next local or parameter */
(34) popstack (locorpar) =
        seth(locorpar,pop(h(locorpar)))
(35) pass (node) = node
```

Primitive Functions

/* These functions perform the primitive functions (ADD, SUB, MINUS, etc.) upon the operands supplied. The operands are expression variables, each of which contains a variable node. The result of the operation upon the specified variable nodes is stored in a node which becomes nested in an expression variable before being returned to the eval routine. */

```
<op> (operand1, operand2) =
        addlevel(define(x|v(x)=(v(*operand1)op v(*operand2)),
                        a(x)=a(*operand1)))
    /* Binary routine evaluates the function and builds a node containing the result of the evaluation. This node is nested within an outer node before being returned.*/
<op> (operand1)
        addlevel(define(x|v(x)=op v(*operand1),
                        a(x)=a(*operand1)))
    /* Unary routine evaluates the function and builds a node containing the result of the evaluation. This node is nested within an outer node before being returned. */
```

CHAPTER III

EXTENSION 1

(ESCAPE MECHANISMS)

1. SIMPL-T Syntax

This section of the paper describes extensions to the base model of SIMPL-T which reflect the addition of escape mechanisms to the language. These mechanisms, namely the EXIT, RETURN, and ABORT statements, affect the program's flow of control and allow the user to cause the early termination of a WHILE loop, a procedure, a function, or the program itself when his pre-defined conditions are met.

The EXIT statement provides a means of escaping from a WHILE loop. In its unlabelled form, it causes the termination of the innermost WHILE loop containing the EXIT statement. By specifying its label in the EXIT statement, however, any one of the loops within which the EXIT is nested may be terminated. Normal execution proceeds upon termination of the proper loop.

The RETURN statement causes a return to the calling procedure or function. In the extended version of SIMPL-T, the RETURN may appear anywhere within the statement list of the segment and need not be the last statement.

The ABORT statement causes the immediate termination of the entire SIMPL-T program.

2. H-Graph Syntax

The changes to the control structure of the language described above, will affect only the control statements of the SIMPL-T model - the remainder of the model is virtually unchanged. The control functions will

in effect ignore the remaining statements within the WHILE loop(s) after an EXIT statement has been encountered, and will ignore the remaining statements within the segment after a RETURN statement has been encountered. For the case of an ABORT, all remaining statements in the program are ignored, and execution of the program terminates.

The modifications and additions to the base model of SIMPL-T follow. When much of a routine is unchanged from that presented in Chapter II, the notation '...' is used to denote the unchanged portion. That portion of a routine which has been changed is designated by a vertical line | .

Syntax changes and additions are preceded by the corresponding line number of Chapter II (if it exists) and the suffix .x . This section uses the suffix .1 . Changes and additions to the semantic functions are handled in a similar manner.

SYNTAX: EXIT, RETURN, ABORT

Statements

S2.1 <stat> :: = <if stmt>|<while stmt>|<case stmt>|<assign stmt>|
<call stmt>|<exit stmt>|<return stmt>|<abort stmt>

S10.1 <while stmt> :: = [<convert predicate>* 1 <stat list> 0 <end>];

class='program',label=<labelname>,stmttype='while'

/* WHILE statements can be labelled */

S21.1 <exit stmt> :: = [<exit operation>];class='operation',

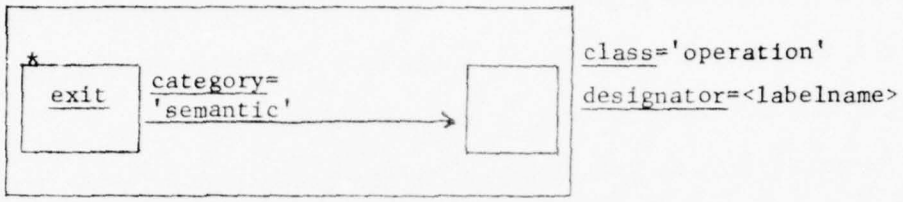
designator=<labelname>

S22.1 <exit operation> :: = exit( )

/* shorthand notation for [exit] → [ ] */

example

<exit stmt>



S23.1 <labelname> :: = '<identifier>'|'0'

S24.1 <abort stmt> :: = [abort( )];class='operation'

F3.1 <semantic function> :: = . . .|exit|abort

3. Semantic Functions

CONTROL FUNCTIONS

/* Traverse is a recursive procedure which controls the traversal through a graph. It calls for the execution of the node 'current node' (via the execute function) and uses its value to calculate the next node in the graph being traversed (via the nextnode function). Traverse then examines the new 'current node'. If it is a data node, we know that the previous node had been the last executable node (see line (c) of nextnode). The new node is returned and the traverse function terminates at this point.

If the new node indicates, by way of a message generated by the execution of the previous node, that an EXIT statement had been encountered (i.e., if ν(current node)=exit), traverse checks to see if 'nodeset' corresponds to the WHILE loop which is to be terminated. If it does not, the EXIT message is passed along until traverse succeeds in finding the proper WHILE loop. When the label of 'nodeset' corresponds to the label of a WHILE loop, a dummy node is returned and the message is in effect

erased.

If the new node indicates that a procedure RETURN statement had been encountered (i.e., if ν(current node)=return), traverse then checks to see if 'nodeset' represents a procedure. If it does not, the RETURN message is passed along until traverse succeeds in finding the innermost procedure. When the procedure is found, a dummy node is returned, and the RETURN message is in effect erased. If the new node indicates that a function RETURN statement had been encountered, (i.e., ν(current node)= freturn), the value of the RETURN expression is stored in the graph of the function definition. The node containing the stored evaluated expression is returned and the RETURN message is in effect erased.

If the new node indicates that an ABORT statement had been encountered (i.e., if ν(current node)=abort), the ABORT message remains unchanged and the execution of the program terminates.

If none of these above conditions has been met, traverse calls itself recursively to calculate the next node in the path and then subjects the new 'current node' to the checks described above. */

(2.1) traverse (current node, nodeset) =

$$
\begin{aligned}
&\{\ \text{class(current node)='data'} \\
&\vee\ \nu\text{(current node)=return}\wedge(\text{segtype(nodeset)='proc'} \vee \\
&\qquad (\exists n \exists x\ \ni \text{nodeset} \in \text{nodes}(h^n(x)) \wedge h^n(x)=h^{n-1}(h(x)) \wedge \\
&\qquad\quad \text{segtype}(x)=\text{'proc'})) \\
&\vee\ \nu\text{(current node)=freturn}\wedge((\text{segtype(nodset)='func'} \wedge \\
&\qquad \text{type}(x \mid \text{role}(x)=\text{'return'} \wedge x \in \text{nodes}(h(\text{nodeset})))= \\
&\qquad \text{type}(*\text{current node}) \wedge (\text{type(nodeset)}= \\
&\qquad \text{type}(*\text{current node})) \\
&\qquad\qquad \vee\ (\exists n \exists x\ \ni \text{nodeset} \in \text{nodes}(h^n(x)) \wedge \\
&\qquad\qquad h^n(x)=h^{n-1}(h(x)) \wedge \text{segtype}(x)=\text{'func'}))
\end{aligned}
$$

```
∨ ν(current node)=exit
∨ ν(current node)=abort
∨  class(current node)='operation'
∨  class(current node)='program'

class(current node)='data' ⇒ current node /*end of traversal*/
ν(current node)=return ⇒              /*procedure RETURN*/
     segtype(nodeset)='proc' ⇒ define(x|ν(x)=true)
      T                      ⇒ current node /*pass on message,
                                   continue search for procedure*/
ν(current node)=freturn ⇒       /*function RETURN*/
     segtype(nodeset)='func' ⇒
       assign(x|role(x)='return'∧ x ∈ nodes(h(nodeset)),
                          current node)
       /* store value of RETURN expression in graph of the
          function */
     T ⇒ current node  /*pass on message, continue search for
                        function*/
ν(current node)=exit ⇒
     stmttype(nodeset)='while' ⇒
       label(nodeset)=designator(current node) ∨
            designator(current node)='0' ⇒ define(x|ν(x)=true)
                  /*proper WHILE loop has been found, erase
                    message*/
     T    ⇒ current node   /*pass on message, continue search
                            for WHILE*/
ν(current node)=abort ⇒ current node /*pass on message*/
```

```
class(current node)='operation' ∨ class(current node)='program'
    ⇒ traverse(nextnode(execute(current node),current node,
          nodeset),nodeset)
    /* continue traversal */
```

/* Nextnode examines the previous node ('lastnode') and the node containing the value resulting from the previous node's execution ('lastnodevalue'). If the previous node had been a RETURN, EXIT, or ABORT statement (line b), a message to this effect is sent back to traverse. If the previous node had been a terminal node (line c), the traversal of the graph is complete, and the previous node is returned to the traverse function. If the previous node executed had had one arc directed to the nextnode (line d), this new node is returned to traverse. If, however, more than one arc had emanated from the last node executed, the arc label must be matched against the value resulting from the execution of the previous node (line e,f). When a unique match is found, the next node to be examined is returned to traverse. */

(3.1) nextnode (lastnodevalue, lastnode, nodeset) =

```
ν(lastnodevalue)=undefined ∨
 ν(lastnodevalue)=return ∨ ν(lastnodevalue)=freturn ∨
 ν(lastnodevalue)=exit ∨ ν(lastnodevalue)=abort ∨
 0 ≤ size(padj(lastnode,h(nodeset))) ≤ 1 /*0 or 1 successors*/ ∨
(((∃x) x∈padj(lastnode,h(nodeset))∧e(lastnode,x)=ν(lastnodevalue)
                                   h(nodeset)
      /* matching arc label for IF, WHILE, CASE statements*/∨
  (∃x) x ∈padj(lastnode,h(nodeset))∧e(lastnode,x)='else')
                                   h(nodeset)
 ∧ size(padj(lastnode,h(nodeset))|e(lastnode,x)=ν(lastnodevalue)
                                   h(nodeset)
          ∨ e(lastnode,x)='else')=1) /*only one match exists*/
             h(nodeset)
```

```
/*  (a)  pass on error message */

        ν(lastnodevalue)=undefined ⇒ lastnodevalue

/*  (b)  pass on return, exit, or abort message */

        ν(lastnodevalue)=return ∨    ν(lastnodevalue)=freturn ∨

        ν(lastnodevalue)=exit ∨      ν(lastnode value)=abort

            ⇒ lastnodevalue

/*  (c) . . .
```

COMPONENT FUNCTIONS

```
(4.1)  eval (node) =

        ≀                                                          ≀
        ≀                    .                                     ≀
        ≀                    .                                     ≀
        ≀                    .                                     ≀
        ≀                    ∨ (category(*node)='semantic' ∧ (. . .∨ ≀
        ≀                        ν(*node)=exit ∨ ν(*node)=abort ...))) ≀
        ≀                                                          ≀
        .                                                          ≀
        .                                                          ≀
        .                                                          ≀

        /* node to be evaluated represents a semantic function */

            category(*node)='semantic' ⇒

                    .
                    .
                    .
                    ν(*node)=exit ⇒ exit(define(x|designator(x)=

                                designator(node)))

                    ν(*node)=abort ⇒ abort([ ])

(11.1)  return (node) =

        ≀ node=[ ] ≀

                    define(x|ν(x)=return)    /* procedure RETURN */

(12.1)  freturn (node) =

        ≀ node=<expr> ≀

                define(x|ν(x)=freturn,h(x)=h(eval(node))) /*function
                                                            RETURN */
```

/* Exit creates a message node which signals that an EXIT statement has been encountered. If no label appeared in the EXIT statement, the designator of the input 'node' would be '0'. If a label did appear in the EXIT statement, the designator of the input 'node' would be the label name. In either case, the value of the node is set to exit to indicate to the control functions (traverse, nextnode) that an EXIT statement has been encountered. */

(36.1) exit (node) =

{ designator(node)=<labelname> }

define (x|designator(x)=designator(node),ν(x)=exit)

/* Abort creates a message node which signals to the control functions (traverse, nextnode) that the program is to be terminated. The input 'node' is empty. */

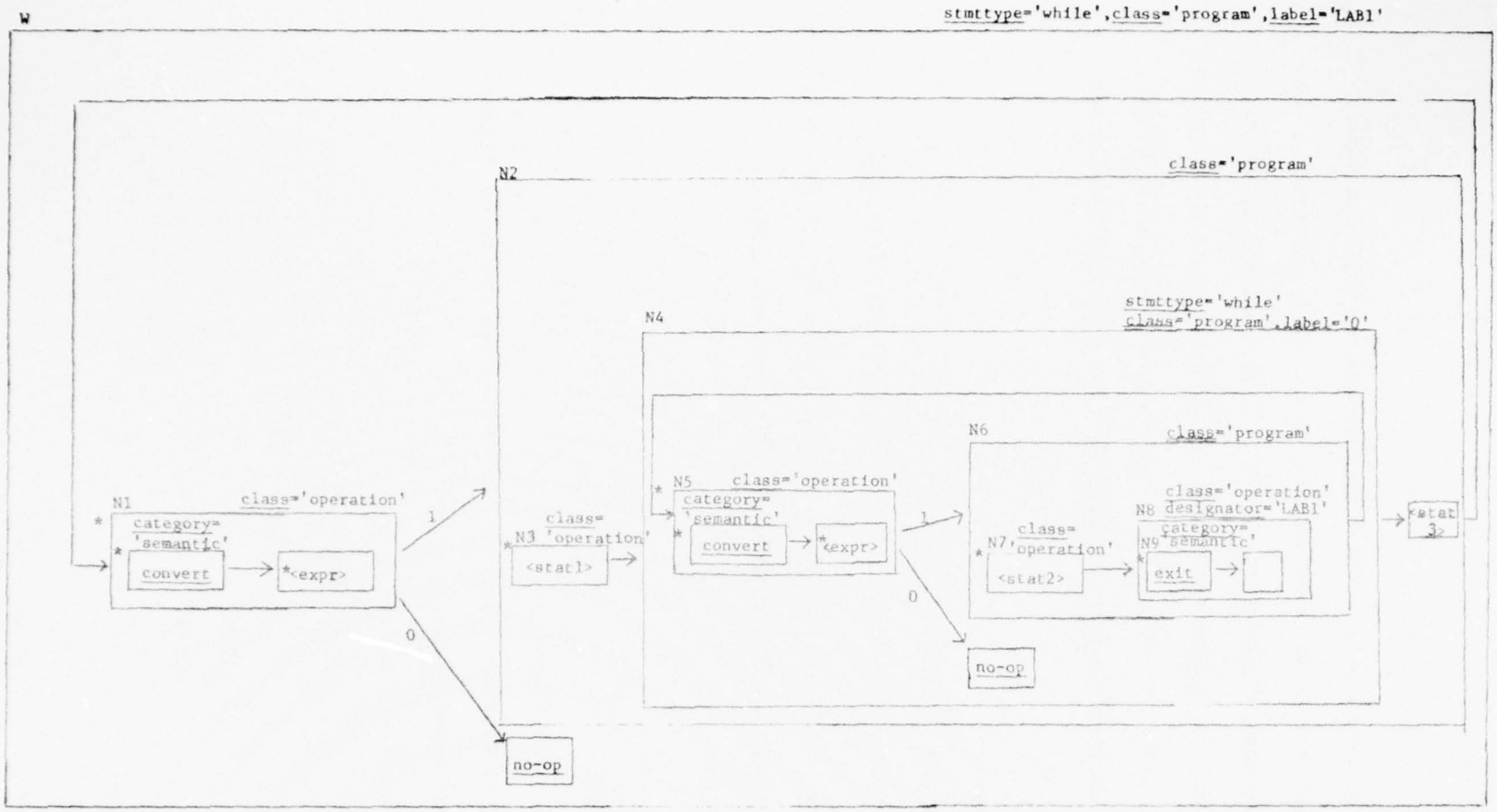
(37.1) abort (node) =

{ node=[ ] }

define (x|ν(x)=abort)

As an example, consider a SIMPL-T statement whose syntax and h-graph representation are given below:

```
while <expr> do <stat1>
                    while <expr> do <stat2>
                                        exit
                     end
                    <stat3>
end
```


W
stmttype='while',class='program',label='LAB1'
N2
class='program'
N4
stmttype='while'
class='program',label='0'
N6
class='program'
N1
class='operation'
category=
'semantic'
convert
*<expr>
1
0
N3
class=
'operation'
<stat1>
N5
class='operation'
category=
'semantic'
convert
*<expr>
1
0
no-op
N7
class=
'operation'
<stat2>
class='operation'
N8 designator='LAB1'
category=
N9 'semantic'
exit
<stat
3>
no-op

The statement W is examined by the execute routine and is found to have class='program'. The nodes of the statement must therefore be traversed, beginning with the conditional expression represented in N1. By assumption, the expression is true (i.e., evaluates to 1). The nextnode routine matches this value 1 against the edge labels emanating from N1 and determines that the next node to be traversed is N2, a statement list. Each statement of the statement list must be examined, beginning with N3 and then proceeding to the WHILE loop represented in N4. The class of N4 is 'program' so it too must be traversed. Its traversal begins with N5, which by assumption is a true expression (i.e., evaluates to 1). The nextnode routine matches this value 1 against the edge labels emanating from N5 and chooses N6 as the next node to be examined. The statement list represented in N6 must be examined one statement at a time, beginning with N7 and then proceeding to the EXIT statement found in N8. Evaluation of the EXIT results in a node whose value is exit and whose designator attribute is 'LAB1'. This node serves as a signal to the control functions that the normally sequential flow of control is being altered. The message is transmitted by the nextnode routine to the traverse routine which checks to see if a WHILE statement has been completed. Toward this end, first N6 and then N4 are examined. A WHILE statement is indeed represented in N4 but its label does not agree with the designator of the EXIT statement and so the search for the proper WHILE statement must continue. The traverse routine examines N2 and then W. A WHILE statement (W) whose label matches the designator of the EXIT statement has been found. Execution of statement W is complete.

The execution of the statement is described below. The following shorthand notation is used:

(1) refers to a node whose value is 1

EXIT node refers to a node where v(node)=exit and

designator(node)=<labelname>

ex=execute

tr=traverse

nn=nextnode

It is assumed that both instances of <expr> in the example are true.

ex(W)=tr(N1,W)

=tr(nn(ex(N1),N1,W),W)

=tr(nn(eval(N1),N1,W),W)

=tr(nn((1),N1,W),W)

=tr(N2,W)

=tr(nn(ex(N2),N2,W)W)

tr(N3,N2)

tr(nn(ex(N3),N3,N2),N2)

eval(N3)

tr(N4,N2)

tr(nn(ex(N4),N4,N2),N2)

tr(N5,N4)

tr(nn(ex(N5),N5,N4),N4)

eval(N5)

tr(nn((1),N5,N4),N4)

tr(N6,N4)

tr(nn(ex(N6),N6,N4),N4)

tr(N7,N6)

tr(nn(ex(N7),N7,N6),N6)

eval(N7)

tr(nn(tr(nn(tr(nn(tr(N8,N6),N6,N4),N4),N4,N2),N2),N2,W),W)

tr(nn(ex(N8),N8,N6),N6)

eval(N8)

tr(nn(EXIT node,N8,N6),N6)

tr(EXIT node,N6)

tr(nn(EXIT node,N6,N4),N4)

tr(EXIT node,N4)

tr(nn(EXIT node,N4,N2),N2)

/*labels do not match*/

tr(EXIT node,N2)

=tr(nn(EXIT node,N2,W),W)

=tr(EXIT node,W)

=true

/*labels match*/

CHAPTER IV

EXTENSION 2

(STRINGS (I))

1. SIMPL-T Syntax

In the basic version of the SIMPL-T language described above, the only data type recognized was 'integer'. An extended version of the language which includes 'string' data types will now be described.

A string is a finite sequence of characters. In this first version of string extensions to SIMPL-T, this sequence of characters will be treated as a unit whose value is the value of the string. No attempt will be made to consider the characters as individual units of information.

A string has two lengths associated with it: a maximum length which reflects the maximum length declared by the user, and a current length which reflects the actual number of characters in the string at a particular point in time.

Strings may be linked together to form a string array. All elements of this array must have the same maximum length. Thus a string array declaration includes two static length declarations: one specifying the maximum size of a string within the array, and the other specifying the number of elements (i.e., strings) in the array.

Binary relational operators can be used in conjunction with string operands, as can the two new operators that are introduced in this section: concatenate and substring. The concatenate operator con generates a string by joining together two existing operand strings. The substring

operator generates a string by extracting a substring from an existing string.

The assignment statement for strings assigns the value of a string expression to a string variable. It is also possible, by using the substring operator on the left side of the assignment statement, to assign a value to a portion (i.e., substring) of a string operand. This facility is called substring assignment.

A string function is a function whose value is a string. The rules governing the use of string functions are analogous to the rules for integer functions described in the basic version of SIMPL-T. Similarly, the conventions followed for passing strings and string arrays as arguments to user procedures and functions are the same as for integers and integer arrays.

The extended version of SIMPL-T defines several intrinsic functions that facilitate programming with strings. Two of these functions are modeled in this paper: trim (which truncates trailing blanks) and digits (which checks to see if each character is a digit).

For a more detailed description of strings, see SIMPL-T manual [3] (pp. 25-39).

2. H-Graph Syntax

The h-graph syntax of the extended model of SIMPL-T has been expanded to include string as well as integer variables. The string variable is represented as a node whose value is the value of the string. Two lengths, a maximum and a current length, are associated with the node. Being that the maximum length remains constant throughout the program, it is assigned to an attribute (maxsize) of the node. The current length, however, changes dynamically during execution and has there-

fore been assigned to a node within the string.

String variables may be linked together in list form to create a one-dimensional string array. This array is represented by a node whose graph consists of nodes representing the individual strings of the array. The number of elements in the array (arraysize), together with other identifying characteristics are reflected in the attributes associated with the array. At the same time, each string of the array retains its own value, list structure, and attributes.

Simple string variables are joined together in a list structure to form a recursive string variable; simple string arrays are joined together to form a recursive string array. These additional forms of data are necessary for the implementation of procedures and functions.

The addition of string functions has been made possible merely by expanding the definition of <typename> to include 'string'. The general rules governing the use of functions need not be modified.

The flow of control in the expanded model, together with its associated control functions, remain unchanged by the addition of strings. The component functions, however, must be adapted to recognize this new data type.

Since integer operands may not be used in conjunction with the two new operators introduced in this section, con and substring, type checking must be performed at compile time to enable the proper evaluation by the eval routine. The assign routine as well assumes extensive type checking upon its parameters at compile time to ensure that strings and integers are handled properly. In addition, the assign routine has been expanded to allow for substring assignment.

The extensions to the base model of SIMPL-T follow.

SYNTAX

Data

Variables

V2.2 <simple var> :: = <integer var>|<string var>

V6.2 <string var> :: = [<length>*→<string>];class='data',type='string',
structure='scalar',form='simple',maxsize=<integer>

V7.2 <string> :: = <charlist> | [ ]

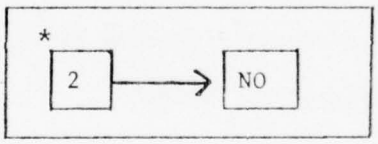
V8.2 <charlist> :: = [any sequence of legal characters]

V9.2 <length> :: = [<integer>]

example

<string var>

string FOUND[3] = 'NO'



class='data',
type='string',
structure='scalar',
form='simple',
maxsize=3

/* The rules governing the passing of string parameters are analogous to those for passing integer parameters. Local variables and parameter variables are defined as recursive variables containing their own stacks. These stacks are defined by a list structure. Recursive string variables and recursive string arrays are now defined:

V4.2 <rec var> :: = <rec integer>|<rec string>

V10.2 <rec string> :: = [<string var>*{→<string var>}$_{o}^{n}$]; class='data',
type='string',structure='scalar',form='rec'

Arrays

A2.2 <simple array> :: = <integer array>|<string array>

A6.2 <string array> :: = [<string var>*{→<string var>}$_{o}^{n}$];class='data',

type='string',structure='array',form='simple',arraysize=

nsize(h(<string array>))

A4.2 <rec array> :: = <rec integer array>|<rec string array>

A7.2 <rec string array> :: = [<string array>*→<string array>$\}_{o}^{n}$];

class='data',type='string',structure='array',form='rec',

arraysize=nsize(h(<rec string array>))

/* The syntax for <expr> and <operation> is unchanged from that defined in previous sections. The syntax for <operator> has been extended. */

E1 <expr> :: = <operation>|<ref predicate>|[<variable>];

class='exprvariable'

Operations

01 <operation>:: = <primitive operation>|<user operation>

02 <primitive operation> :: = [<operator>*→<operand list>];

class='operation'

05.2 <operator> :: = <primitive unary function>;form='unary',

category='primitive'|

<primitive binary function>;form='binary',

category='primitive'|

<primitive tertiary function>;form='tertiary',

category='primitive'|

<primitive quint function>;form='quint',

category='primitive'

010.2 <primitive unary function> :: = ...|<unary string function>;

optype='string'

018.2 <unary string function> :: = [trim]|[digits]

06.2 <primitive binary function> :: = ...|<binary string function>;

optype='string'

O19.2 <binary string function> :: = [con]

O20.2 <primitive tertiary function> :: = <tertiary string function>;
optype='string'

O21.2 <tertiary string function> :: = [substringrtn]|[substring]

O22.2 <primitive quint function> :: = <quint string function>;
optype='string'

O23.2 <quint string function> :: = [substringassign]

Program Structure

P6.2 <typename> :: = 'integer'|'string'

S13.2 <℘> :: = <variable>|<ref predicate>|<substring predicate>

S25.2 <substring predicate> :: = [<substring operation>];class='operation'

S26.2 <substring operation> :: = substring(<expr>,<expr>,<expr>)

Semantic Operations

F3.2 <semantic function> :: ... [stringeval]|[substringeval]

3. Semantic Functions

COMPONENT FUNCTIONS

(4.2) eval (node) =

```
class(node)='data'∧(form(node)='rec' ∨ form(node)='simple')∨
class(node)='exprvariable'∨
class(node)='operation'∧((category(*node)='primitive'∧
                ∧((form(*node)='binary' ∧
                  (((optype(*node)='integer' ∨ optype(*node)=
                     'rel' ∨ optype(*node)='log')
                     ∧ integerexpr(oper1(node))=1
                     ∧ integerexpr(oper2(node))=1)
                  ∨ ((optype(*node)='string' ∨ optype(*node)=
                     'rel')
```

```
                ∧ stringexpr(oper1(node))=1
                ∧ stringexpr(oper2(node))=1)))
            ∨(form(*node)='unary'
              ∧(((optype(*node)='integer' ∨ optype(*node)=
                'rel') ∧ integerexpr(oper1(node))=1)
              ∨ (optype(*node)='string'
                ∧ stringexpr(oper1(node))=1)))
            ∨ (form(*node)='tertiary'
              ∧ optype(*node)='string'
                ∧ stringexpr(oper1(node))=1
                ∧ integerexpr(oper2(node))=1
                ∧ integerexpr(oper3(node))=1)
            ∨ (form(*node)='quint'
              ∧ optype(*node)='string'
                ∧ stringexpr(oper1(node))=1
                ∧ stringexpr(oper2(node))=1
                ∧ integerexpr(oper3(node))=1
                ∧ integerexpr(oper4(node))=1
                ∧ integerexpr(oper5(node))=1)))
                  ∨ category(*node)='function'
                  ∨ (category(*node)='semantic'
                      ∧ (∨(*node)=assign ∨ . . .
                      ∨  (*node)=stringeval
                      ∨  (*node)=substringeval)))
        .
        .
        .
        /* node to be evaluated represents an operation*/
class(node)='operation' ⇒
  category(*node)='primitive' ⇒
    form(*node)='binary' ⇒
```

```
                    ν(*node)(eval(oper1(node)),eval(oper2(node)))
                /* where if  ν(*node)=add ⇒ add(eval(oper1
                                (node)),eval(oper2(node)))
                          ν(*node)=sub ⇒ sub(eval(oper1
                                (node)),eval(oper2(node)))
                          ν(*node)=con ⇒ con(eval(oper1
                                (node)),eval(oper2(node))) etc.*/
            form(*node)='unary' ⇒
                ν(*node)(eval(oper1(node)))
            form(*node)='tertiary' ⇒
                ν(*node(oper1(node),oper2(node),oper3(node))
            /* where if  ν(*node)=substringrtn ⇒ substringrtn
                    (oper1(node),oper2(node),oper3(node))*/
            form(*node)='quint' ⇒
                ν(*node)(oper1(node),oper2(node),oper3(node),
                        oper4(node),oper5(node))
        .
        .
        .
    /* node to be evaluated represents a semantic function */
        category(*node)='semantic' ⇒
                ν(*node)=assign ⇒ (oper1(node),oper2(node))
                .
                .
                .
                ν(*node)=stringeval(oper1(node),oper2(node))
                ν(*node)=substringeval(oper1(node),oper2(node),
                        oper3(node),oper4(node))
```

/* The assign routine, together with stringeval and substringeval, evaluate the right side ('rp') of the statement and assign its value to the node represented by 'lp'. Being that 'rp' designates an expression, its

nested node's value is used in the assignment. The routine is written in modular fashion to avoid the possibility of side effects. Type checking is performed to ensure that strings are not assigned to integer variables and integers are not assigned to string variables. */

(7.2) assign (ℓp, rp) =

{ ((integervariable(ℓp)=1 ∨ integerarrayelement(ℓp)=1)
∧ integerexpr(rp)=1)
∨((stringvariable(ℓp)=1 ∨ stringarrayelement(ℓp)=1)
∧ stringexpr(rp)=1)
∨(class(ℓp)='operation' ∧ ν(*ℓp)=substring ∧ stringexpr(rp)=1)) }

/* assignment to integer variable or integer array element by an integer expression */

integervariable(ℓp)=1 ⇒ setv(eval(ℓp),ν(*eval(rp)))

integerarrayelement(ℓp)=1 ⇒ setv(*eval(ℓp),ν(*eval(rp)))

/* assignment to string variable or string array element by a string expression */

stringvariable(ℓp)=1 ⇒ stringeval(eval(ℓp),eval(rp))

stringarrayelement(ℓp)=1 ⇒ stringeval(*eval(ℓp),eval(rp))

/* assignment to a substring of the string variable specified in the left part ('ℓp') of the statement by the expression specified in the right part ('rp') of the statement.

The string variable specified in the 'ℓp' of the statement is represented by oper1(ℓp). The field that specifies which character of the string is to serve as the beginning of the substring is represented by oper2(ℓp) . The length of the substring is represented by oper3(ℓp) .

An additional routine substringeval is needed so that

each of the parameters is evaluated only once, thereby eliminating the possibility of side effects. The parameters of the substring operation appearing on the left side of the statement, along with the expression appearing on the right hand side are evaluated and passed to substringeval for further processing. If the field specifying the length of the substring (oper3(ℓp)) is omitted, it is assumed that the substring is to consist of all characters of the original string beginning at the location specified by oper2(ℓp) . The empty node is then passed along to fill the place of the omitted parameter. All parameters passed to substringeval (except for the empty node) are expression variables. */

```
class(ℓp)='operation' ∧ ν(*ℓp)=substring ⇒
    size(padj(oper2(ℓp),h(elt(padj(*ℓp,h(ℓp))))))=0
                                    /* oper3(ℓp) omitted*/
      ⇒ substringeval(oper1(ℓp),eval(oper2(ℓp)),
                    define,eval(rp))
    size(padj(oper2(ℓp),h(elt(padj(*ℓp),h(ℓp))))) > 0
                    /* oper3(ℓp) included in substring
                                    operation */
      ⇒ substringeval(oper1(ℓp),eval(oper2(ℓp)),
                    eval(oper3(ℓp)),eval(rp))
```

/* Stringeval assigns the string constructed by substringrtn to the string variable 'ℓp'. Substringrtn is treated as a primitive operation which creates a copy of the string 'rp', truncated to the maximum length of 'ℓp', if necessary. The parameters passed to substringrtn are expression variables which are defined as follows:

parameter 1) the string to be copied,

parameter 2) the starting location of the string to be copied ($\equiv$ 1), and

parameter 3) the length of the string to be constructed.

Substringrtn returns a string nested within an expression variable. The graph of the new string is assigned to '&p' and the stringeval routine, as well as the assign routine, are complete.

(The string construction is described in detail in Chapter V where substringrtn is treated as a semantic function (46.3).) */

(38.2) stringeval (&p, rp) =

$$\left\{\begin{array}{l} \text{class(rp)='exprvariable'} \wedge \text{type(*rp)='string'} \\ \wedge\ \text{class(\&p)='data'} \wedge \text{type(\&p)='string'} \end{array}\right\}$$

/* length of 'rp' does not exceed maximum length of '&p' */

maxsize(&p) $\geq$ ν(**rp) $\Rightarrow$

seth(&p,h(*substringrtn(rp,addlevel(define(x|ν(x)=1,

type(x)='integer')),addlevel(define(x|ν(x)=

ν(**rp),type(x)='integer')))))

/* length of 'rp' exceeds maximum length of 'lp' */

maxsize(&p) $<$ ν(**rp) $\Rightarrow$

seth(&p,h(*substringrtn(rp,addlevel(define(x|ν(x)=1,

type(x)='integer')),addlevel(define(x|ν(x)=

maxsize(&p),type(x)='integer')))))

/* Substringeval is referenced by the assign routine for the purpose of assigning the string variable 'rp' to a string designated by the substring operator. The first three parameters of the substringeval routine are expression variables which furnish information relating to the substring operation, namely:

string 1 = original string operand

var1 = location of first character within string to be assigned (F1)

var2 = number of characters to be assigned (F2)

The fourth parameter 'rp' is an expression variable which represents the string variable that is to be assigned to the substring appearing on the left side of the assignment statement.

If var2 is empty, indicating that the length field was omitted, the length of the substring is calculated and the substringassign routine is then invoked. This routine, which performs the actual assignment, shall be treated as a primitive operation in this section of the paper. The parameters passed to the routine are defined as follows:

parameter 1) expression variable containing original string

parameter 2) expression variable containing the string variable to be assigned to the substring

parameter 3) expression variable containing the location within the original string at which the assignment will begin

parameter 4) expression variable containing the length of the substring to be assigned

parameter 5) a node whose value is initially set to 1.

Upon returning from substringassign, both the substringeval and assign routines are complete.

(The substring assignment is described in detail in Chapter V where substringassign is treated as a semantic function (48.3).) */

(39.2) substringeval (string1, var1, var2, rp) =

{ class(string1)='exprvariable' ∧ class(var1)='exprvariable' ∧ }

```
{ (class(var2)='exprvariable' ∨ var2=[ ]) ∧ class(rp)=
                    'exprvariable' ∧ type(*rp)='string' ∧
  substringsyntax(*stringl,*var1,*var2)=1
                          /* substring syntax OK */ }

ν(*var1) > 0 ⇒                    /* F1 positive */
   ν(*var2)=0 ⇒ define(x|ν(x)=true)   /*F2=0, no action taken*/
   nsize(h(var2))=0 ∧ ν(*var1) ≤ ν(**stringl)
                    /* F2 omitted and F1 falls within bounds of string*/
/* F2 omitted */   ⇒ substringassign(stringl,rp,var1,addlevel
                    (define(x|ν(x)=ν(**stringl)-ν(*var1)+1,type(x)=
                    'integer')),define(x|ν(x)=1,type(x)='integer'))
   ν(*var2) ≥ 0 ∧  ν(*var1)+ν(*var2)-1 ≤ ν(**stringl)
                    /* F2 positive and lies within bounds of string*/
/* F2 positive */  ⇒ substringassign(stringl,rp,var1,var2,define(x|ν(x)=1,
                    type(x)='integer'))
   T ⇒ define(x|ν(x)=undefined,class(x)='data')
                                    /* error condition */
   T ⇒ define(x|ν(x)=undefined,class(x)='data')
                                    /* error condition */
```

/* Valuenode returns a node whose value is equal to that of an evaluated argument node. */

(20.2) valuenode (node) =

```
{ class(node)='exprvariable' ∧ (type(*node)='integer' ∨
          type(*node)='string') ∨
  class(node)='data' ∧ (type(node)='integer' ∨
          type(node)='string') }
```

```
        class(node)='exprvariable' ⇒
            type(*node)='integer' ⇒ define(x|ν(x)=ν(*node))
            type(*node)='string' ⇒ assign(define(x|type(x)='string',
                        class(x)='data',structure(x)='scalar',
                        form(x)='simple',maxsize(x)=maxsize(*node)),node)
        class(node)='data' ⇒
            type(node)='integer' ⇒ define(x|ν(x)=ν(node))
            type(node)='string' ⇒ valuenode(addlevel(node))
/*  Pushloc adds a local variable to its corresponding stack */
(28.2)  pushloc (local) =
          ≀ (type(local)='integer' ∨ type(local)='string')    ≀
          ≀ ∧ structure(local)='scalar' ∧ form(local)='rec'   ≀
        type(local)='integer' ⇒
            seth(local,push(h(local),define(x|a(x)=a(*local))))
        type(local)='string' ⇒
            seth(local,push(h(local),define(x|a(x)=a(*local),h(x)=
                append(list(define(y|ν(y)=0)),define))))
(29.2)  installocarray (array, count, stack) =
          ≀ (type(stack)='integer' ∨ type(stack)='string') ∧ ≀
          ≀       form(stack)='rec'                          ≀
            ν(count)=0 ⇒  /* array ready to be added to stack */
                locattr(seth(stack,push(h(stack),array)))
            ν(count)≠0 ⇒  /* array building still in progress */
                type(stack)='string' ⇒
                    installocarray(seth(array,push(h(array),
                            define(y|a(y)=a(**stack),h(y)=append
                            (list(define(x|ν(x)=0)),define)))),
```

```
                setv(count,v(count)-1),
                stack)
                /* proceed to install next element of
                   array */
        type(stack) = 'integer' ⇒
                .
                .
                .
```

/* This routine returns the value 1 if 'node' is a string variable */

(40.2) stringvariable (node) =

```
        structure(node)='scalar' ∧ type(node)='string' ⇒ 1
        T                                              ⇒ 0
```

/* This routine returns the value 1 if 'node' is a string array element */

(41.2) stringarrayelement (node) =

```
        class(node)='operation' ∧ v(*node)=ref ∧ type(oper1(node))=
                                        'string' ⇒ 1
        T                                        ⇒ 0
```

/* This routine returns the value 1 if the syntax of the substring is valid, i.e., 'node' is of type string, F1 and F2 are integer expressions, F2 may be omitted. */

(42.2) substringsyntax (node, F1, F2) =

```
        type(node)='string' ∧ integerexpr(F1)=1
      ∧ (integerexpr(F2)=1 ∨ nsize(h(F2))=0) ⇒ 1
        T                                    ⇒ 0
```

/* This routine returns the value 1 if 'expr' is a string expression */

(43.2) stringexpr (expr) =

```
        (class(expr)='operation' ∧ optype(*expr)='string')
```

```
V type(*expr)='string' ⇒ 1
  T                    ⇒ 0
```

CHAPTER V

EXTENSION 3

(STRINGS (II))

In this section an added level of detail will be introduced into the modeling of strings. The <charlist> which heretofore has been treated as a unit, will now be treated as a node which itself is composed of individual nodes of information. Each of these nodes represents one character of the string. A list structure is used to describe this sequence of characters.

Because of the added level of detail, the implementation of routines that were called in the previous section can now be described. The list construction primitives push and append are used by these routines to generate new strings, whereas the list accessing primitives item and next are used to address individual characters within an already existing string.

1. H-Graph Syntax

Data

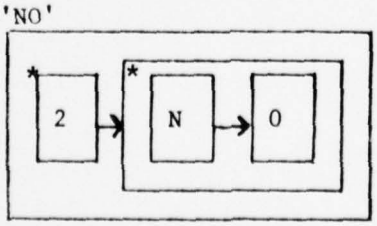
V6.3 <string var> :: = [<length>*→<string>]; class='data',type='string',
structure='scalar',form='simple',maxsize=
nsize(h(<charlist>))

V8.3 <charlist> :: = $[\langle character\rangle * \{\rightarrow \langle character\rangle\}_{o}^{n}]$

example

<string var>

string FOUND[3] ='NO'



class='data',
type='string'
structure='scalar'
form='simple'
maxsize=3

Operations

/* The primitive operations introduced in the previous section will now be treated as semantic functions. The additions of the previous section pertaining to primitive operations can now be deleted. */

Semantic Operations

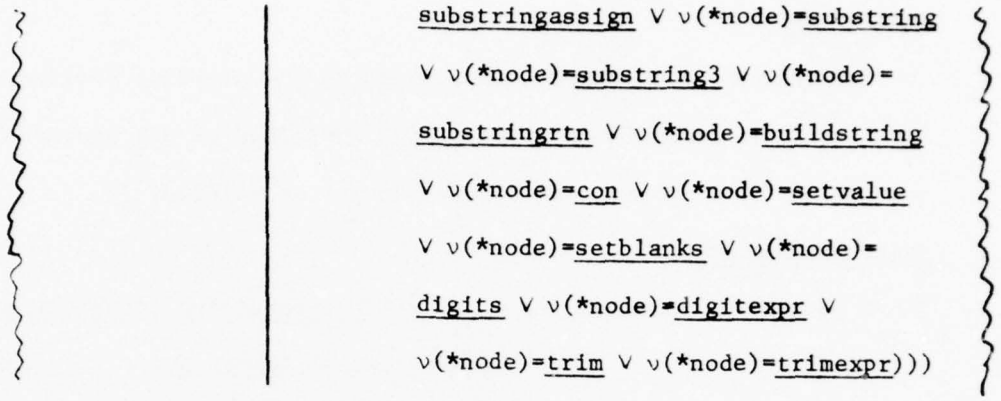
F3.3 <semantic function> :: = ... |[substringassign]|[substring]|
[substring3]|[substringrtn]|[buildstring]|
[setvalue]|[setblanks]|[trim]|[trimexpr]|
[digits]|[digitexpr]|[con]

2. Semantic Functions

COMPONENT FUNCTIONS

(4.3) eval (node) =

class(node)='data' ∧ (form(node)='rec' ∨ form(node)='simple') ∨
class(node)='exprvariable' ∨
class(node)='operation' ∧ ((category(*node)='primitive'
∧ ((form(*node)='binary' ∧ (optype(*node)=
'integer' ∨ optype(*node)='rel'
∨ optype(*node)='log') ∧ integerexpr
(oper1(node))=1 ∧ integerexpr(oper2(node))=1)
∨ (form(*node)='unary' ∧ (optype(*node)=
'integer' ∨ optype(*node)='log')
∧ integerexpr(oper1(node))=1)))
∨ category(*node)'function'
∨ (category(*node)='semantic' ∧
(ν(*node)=assign ∨ ... ∨ ν(*node)=
stringeval ∨ ν(*node)=
substringeval ∨ ν(*node)=




```
.
.
.
/*  node to be evaluated represents an operation */
     class(node)='operation' ⇒
       category(*node)='primitive' ⇒
          form(*node)='binary' ⇒
             ν(*node)(eval(oper1(node)),eval(oper2(node)))
         /* where if ν(*node)=add ⇒ add(eval(oper1(node)),eval(oper2(node)))
                     ν(*node)=sub ⇒ sub(eval(oper1(node)),
                        eval(oper2(node))) */
          form(*node)='unary' ⇒
             ν(*node)(eval(oper1(node)))
/*  node to be evaluated represents a function call */
       category(*node)='function' ...
/*  node to be evaluated represents a semantic function */
       category(*node)='semantic' ⇒
         .
         .
         .
       | ν(*node)=substringassign ⇒ substringassign(oper1(node),
       |              oper2(node), ... , oper5(node))
```

```
v(*node)=substring ⇒ substring(oper1(node),oper2(node),
                oper3(node))
v(*node)=substring3 ⇒ substring3(oper1(node),oper2(node),
                oper3(node))
v(*node)=substringrtn ⇒ substringrtn(oper1(node),
                oper2(node),oper3(node))
v(*node)=buildstring ⇒ buildstring(oper1(node),
                oper2(node),...,oper6(node))
v(*node)=con ⇒ con(oper1(node),oper2(node))
v(*node)=setvalue ⇒ setvalue(oper1(node),
                oper2(node),oper3(node),oper4(node))
v(*node)=setblanks ⇒ setblanks(oper1(node),oper2(node),
                oper3(node))
v(*node)=digits ⇒ digits(oper1(node))
v(*node)=digitexpr ⇒ digitexpr(oper1(node),oper2(node))
v(*node)=trim ⇒ trim(oper1(node))
v(*node)=trimexpr ⇒ trimexpr(oper1(node),oper2(node))
```

/* The substring operator generates a string by extracting a substring of length F2 from its string operand, beginning at character position F1. The string operand in this routine is the node 'stringl'. The two fields F1 and F2 of the substring operator may be any integer variables and are nested in the expression variables 'var1' and 'var2' respectively. The F2 field may be omitted, in which case the substring from character F1 to the end is implied. If F2 is omitted, the substring routine creates a node to simulate the F2 field before calling substring3. */

```
(44.3)  substring (stringl, var1, var2) =
            v(*var1) > 0 ⇒        /* F1 positive */
```

```
/*F2 omitted*/  nsize(h(var2))= 0 ⇒ substring3(stringl,varl,addlevel
                                    (define(x|ν(x)=ν(**stringl)-ν(*varl)+1,
                                    type(x)='integer',class(x)='data',
                                    form(x)='simple')))
/*F2 included*/ nsize(h(var2)) > 0 ⇒ substring3(stringl,varl,var2)
          ν(*varl) ≤ 0 ⇒ define(x|ν(x)=undefined,class(x)='data')
                                                  /*error condition*/
/*  Substring3 returns an expression variable containing the null string
or invokes substringrtn to build the new string, in which case the ex-
pression variable containing the new string is returned. */
(45.3)  substring3 (stringl, varl, var2) =
          substringsyntax(*stringl,*varl,*var2)=1
                                          /* substring syntax OK */
          ν(*varl)+ν(*var2)-1 ≤ ν(**stringl) ⇒  /*substring lies within
                                                 bounds of stringl*/
/*F2≠0*/      ν(*var2) > 0 ⇒ substringrtn(stringl,varl,var2)
                                                /*build new string*/
/*F2=0*/      ν(*var2)= 0 ⇒ addlevel(define(x|class(x)='data',
                            type(x)='string',structure(x)='scalar',
                            form(x)='simple',maxsize(x)=0,h(x)=
                            append(list(define(x|ν(x)=0)),define)))
/*F2 < 0*/    ν(*var2) < 0 ⇒ define(x|ν(x)=undefined,class(x)='data')
                                          /*error condition*/
          ν(*varl)+ν(*var2)-1 > ν(**stringl) ⇒ define(x|ν(x)=undefined,
                            class(x)='data')
            /* error condition - substring does not lie within bounds
               of stringl */
```

/* Substringrtn creates a substring from the node nested in 'stringl', beginning at character position ν(*varl). The length of the substring being created is ν(*var2). The resulting string consists of a length component and a character list component. The string is nested in an expression variable before being returned.

An auxiliary routine buildstring is invoked to generate the list of characters. */

```
(46.3) substringrtn (stringl, varl, var2) =
        ≢ class(stringl)='exprvariable' ∧ class(varl)='exprvariable' ≢
          ∧ class(var2)='exprvariable' ∧
          substringsyntax(*stringl,*varl,*var2)=1
       addlevel(define(z|class(z)='data',type(z)='string',
                                  structure(z)='scalar',form(z)=
                                  'simple',maxsize(z)=ν(*var2),h(z)=
/*length component*/       append(list(define(y|ν(y)=ν(*var2))),
/*character list component*/    buildstring(define,
   /*first character*/          define(x|ν(x)=ν(item(ν(*varl),h(next
                                   (**stringl,h(*stringl))))))),
   /*set count = 0*/            define(x|ν(x)=0),
  /*node containing original
   character list*/             next(**stringl,h(*stringl)),
   /*start location*/           varl,
   /*substring length*/         var2))))
```

/* Buildstring is a recursive routine which builds a node whose graph is a character list. The list contains the characters of the original string ('charstring') beginning at character position ν(*V1) and continuing for ν(*V2) characters. One 'node' at a time is appended to

the list. When the counter indicates that the string is complete, (i.e., when ν(count)= (*V2)) , the node 'stringlist', whose graph is the completed list of characters, is returned. */

```
(47.3) buildstring (stringlist, node, count, charstring, V1, V2) =
        ≀ class(V1)='exprvariable' ∧ class(V2)='exprvariable' ≀
        ν(count)=ν(*V2) ⇒ stringlist          /*string completed*/
        ν(count) < ν(V2) ⇒
            buildstring(seth(stringlist,append(h(stringlist),node)),
               define(x|ν(x)=ν(item(ν(*V1)+ν(count)+1,h(charstring)))),
               setv(count,ν(count)+1),
               charstring,  V1 ,  V2 )
```

/* This routine assigns a value to a substring of a string variable. The input to substringassign is as follows:

left: expression variable containing the string variable undergoing assignment

right: expression variable containing the string variable whose value will be assigned to a substring of 'left'

position: expression variable containing the node whose value is the starting position within the original string variable at which the replacement will begin

length: expression variable containing node whose value corresponds to the number of characters to be replaced, as specified in the substring operation.

index: node whose value is initially set to 1.

Beginning at character position ν(*position) of the original string variable, the first ν(*length) characters of node (*right) are assigned. If ν(*length) exceeds the length of the string in 'right',

the value of the specified substring is extended with trailing blanks so that the replacement of the desired number of characters can be made complete. The remaining characters of the original string ('left') are not changed, nor is the length of 'left' changed. */

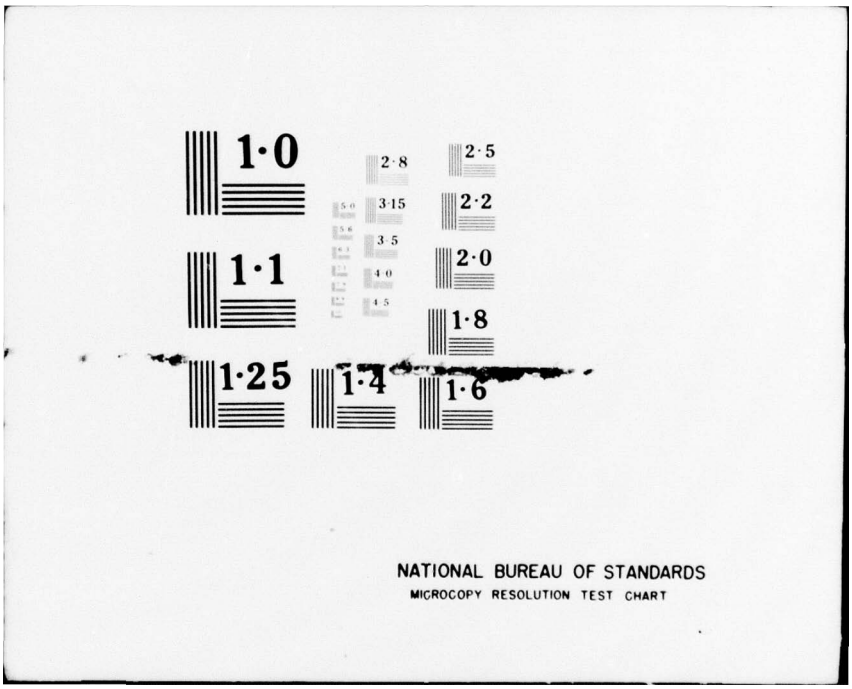
(48.3) substringassign (left, right, position, length, index) =

```
{ class(left)='exprvariable' ∧ type(*left)='string' ∧
  type(*right)='string' ∧ class(right)='exprvariable' ∧
  class(position)='exprvariable' ∧ type(*position)='integer' ∧
  class(length)='exprvariable' ∧ type(*length)='integer'∧
  type(index)='integer' }

ν(*length) ≤ ν(**right) ⇒  /*length of right string exceeds sub-
                              string to be replaced*/
    ν(index) ≤ ν(**right) ⇒  /* assign first (length of left
                                string) characters */
        execute(define(x|class(x)='program',h(x)=append(list(
            [setvalue(left,right,position,index)]),
            [substringassign(left,right,position,length,
                setv(index,ν(index)+1))])))
                              /*proceed to next character*/
    ν(index) > ν(**right) ⇒ define(x|ν(x)=true)
                              /*assignment complete*/

ν(*length) > ν(**right) ⇒  /*length of left substring to be re-
                              placed exceeds length of right string*/
    ν(index) ≤ ν(**right) ⇒  /*assign characters of 'right'
                                to 'left' string*/
        execute(define(x|class(x)='program',h(x)=append(list(
            [setvalue(left,right,position,index)]),
```

AD-A043 380 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
A FAMILY OF H-GRAPH MODELS FOR SIMPL-T.(U)
JUL 77 S J GORLIN
UNCLASSIFIED TR-552 AFOSR-TR-77-0948 AF-AFOSR-3181-77 NL

2 OF 2
AD
A043 380

END
DATE
FILMED
9-77
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

```
                    [substringassign(left,right,position,length,
                        setv(index,v(index)+1))])))
                            /*proceed to next character*/
                v(index) > v(**right) ⇒
                 v(index) ≤ v(*length) ⇒   /* add trailing blanks */
                 execute(define(x|class(x)='program',h(x)=append
                     (list([setblanks(left,position,index)]),
                          [substringassign(left,right,position,length,
                              setv(index,v(index)+1))])))
                              /*proceed to next character*/
                v(index) > v(*length) ⇒ define(x|v(x)=true)
                              /*assignment complete*/
```

/* Setvalue assigns a character from right string to left string beginning at location 'position' */

(49.3) setvalue (left, right, position, index) =

```
        ⎧ type(*left)='string' ∧ class(right)='exprvariable' ∧
        ⎪ type(*right)='string' ∧ class(position)='exprvariable' ∧
        ⎩ type(*position)='integer'
       setv(item(v(*position)+v(index)-1,h(next(**left,h(*left)))),
          v(item(v(index),h(next(**right,h(*right))))))
```

/* Setblanks assigns blanks to left string beginning at location 'position' */

(50.3) setblanks (left, position, index) =

```
        ⎧ type(*left)='string' ∧                                    ⎫
        ⎩ type(*position)='integer'                                 ⎭
       setv(item(v(*position)+v(index)-1,h(next(**left,h(*left)))), b )
```

```
/*  This routine concatenates the nodes nested in 'stringl' and 'string2'
and returns the resulting node nested in an outer node.  The auxiliary
routine buildstring is used to create a copy of the character list of
'stringl'.  The routine is then called a second time to append the char-
acter list of'string2' to the newly created list.  This list is then ap-
pended to the length component to complete the concatenation. */

(51.3)  con (stringl, string2) =

        { class(stringl)='exprvariable' ∧ type(*stringl)='string' ∧ }
        { class(string2)='exprvariable' ∧ type(*string2)='string'  }
              addlevel(define(z|class(z)='data',type(z)='string',
                    structure(z)='scalar',maxsize(z)=maxsize
                    (*stringl)+maxsize(*string2),
                    form(z)='simple',h(z)=
/*length component*/   append(list(define(y|ν(y)=ν(**stringl)+
                       ν(**string2))),
/*character list component*/ buildstring(buildstring(define,define(x|ν(x)=
                                ν(item(1,h(next(**stringl,
/*copy stringl*/                  h(*stringl))))))),define(x|ν(x)=0),
                                next(**stringl,h(*stringl)),
                                addlevel(define(x|ν(x)=1)),
                                addlevel(define(x|ν(x)=
                                ν(**stringl)))),
                          define(x|ν(x)=ν(item(1,h(next(**string2,
                                h(*string2)))))),
/*string2*/                 define(x|ν(x)=0),
                          next(**string2,h(*string2)),
                          addlevel(define(x|ν(x)=1)),
                          addlevel(define(x|ν(x)=ν(**string2)))))))
```

/* Digits returns an expression variable whose inner node has the value 1 if each character in the string is a digit (0-9) and otherwise returns an expression variable whose inner node has the value 0. In order to avoid side effects, the string expression is evaluated only once by the eval routine. The digits routine separates the string variable into its character list and length components. It then calls on digitexpr to perform the actual examination of each of the characters. */

```
(52.3)  digits (string) =
          { class(string)='exprvariable' ∧ type(*string)='string' }
/*null string*/  ν(**string) = 0 ⇒ addlevel(define(x|ν(x)=0)
                  ν(**string) > 0 ⇒ digitexpr(next(**string,h(*string)),
                                              **string)
```

/* The 'charlist' parameter for the digitexpr routine corresponds to the characters of the string under scrutiny. The 'counter' parameter indicates the number of characters that have still not been examined. Its initial value is set to the total number of characters in the string. It is then decremented until all characters have been examined. If each character lies between 0-9, a nested node is returned whose value is 1. */

```
(53.3)  digitexpr (charlist, counter) =
          ν(counter)=0 ⇒ addlevel(define(x|ν(x)=1)) /*only digit found*/
          ν(item(ν(counter),charlist))=0 ∨ ν(item(ν(counter),
                        charlist))=1 ∨
            ... ∨ ν(item(ν(counter),charlist))=9 ⇒
                  digitexpr(charlist,setv(counter,ν(counter)-1))
          T ⇒ addlevel(define(x|ν(x)=0))  /*non-digit found*/
```

/* Trim builds a string which corresponds to the string contained in

the expression variable 'string' truncated to remove trailing blanks. The resulting string is nested in an expression variable before being returned by the trim routine. The original string remains unchanged. To avoid side effects, the string expression is evaluated once by the eval routine. The trim routine separates the string variable into its character list and length components. It then calls on trimexpr to perform the actual truncation. */

```
(54.3)  trim (string) =
          { class(string)='exprvariable' ∧ type(*string)='string' }
          ν(**string)=0 ⇒ string          /* null string */
          ν(**string) > 0 ⇒ trimexpr(next(**string,h(*string)),**string)
```

/* The 'charlist' parameter for the trimexpr routine corresponds to the characters of the string being examined. The 'counter' parameter indicates the number of characters that have not yet been examined. Its initial value is set to the total number of characters in the string. It is then decremented to reflect the number of non-blank characters in 'charlist'. The auxiliary routine buildstring is used to build the truncated character list. */

```
(55.3)  trimexpr (charlist, counter) =
          ν(counter)≠0 ∧ ν(item(ν(counter),charlist))= ⊼  ⇒
              trimexpr(charlist,setv(counter,ν(counter)-1))
                    /* check for blanks and decrement counter */
          T ⇒       /* build string */
                addlevel(define(z|class(z)='data',type(z)='string',
                    structure(z)='scalar',form(z)='simple',maxsize(z)=
                    counter,h(z)=
          append(list(define(y|ν(y)=counter)),
```

```
buildstring(define,
            define(x|ν(x)=ν(item(1,h(charlist)))),
            define(x|ν(x)=0),
            charlist,
            addlevel(define(x|ν(x)=1)),
            addlevel(define(x|ν(x)=counter))))))
```

CONCLUSION

Several h-graph models for SIMPL-T have been presented in this paper. These models attest to the fact that it is possible to extend the original model without destroying its basic framework. The inclusion of string data in Chapters IV, V required additions, but very few changes, to the original component functions of the base model. The control functions presented in Chapter II were easily adapted to provide the flow of control necessitated by the addition of escape mechanisms in Chapter III. The model is cognizant of what has preceded a particular statement, and can therefore EXIT from a WHILE loop or RETURN from a procedure with no major changes to the control functions. Had SIMPL-T included a GOTO statement among its escape mechanisms, however, major revisions to the original model would have been necessary, since the control functions, as presented in these models, have no provision for forward jumps. The basic model presented in this paper did prove sufficiently flexible, however, to include the language features that actually exist in SIMPL-T.

The models themselves are complex, but they merely reflect the complexity of the semantics of the underlying language. They are written in modular fashion so as to preserve the independent character of the extensions. The addition of an EXIT statement, for example, has no effect on the handling of strings, and vice versa. When adding a new feature to the language, the designer need not be concerned with the effects his changes may have on unrelated features of the language. He can rather focus his attention toward the task of incorporating the new features within the existing framework. In this way, he can continue to use modeling as a tool for the further design and modification of the language.

REFERENCES

[1] Basili, V. R., A structured approach to language design, Journal of Computer Languages, 1 (1975), pp. 255-273.

[2] Basili, V. R. and A. J. Turner, A hierarchical machine model for the semantics of programming languages, ACM-IEEE Symposium on High Level Language Computer Architecture, ACM and IEEE (November 1973)

[3] Basili, V. R. and A. J. Turner, SIMPL-T: A Structured Programming Language, University of Maryland, Computer Science Center, Computer Note CN-14.2 (1975).